conference

proceedings

USENIX Fifth Annual Tcl/Tk Workshop '97 Proceedings *Boston, Massachusetts July 1997*

The Fifth Annual Tcl/Tk Workshop '97 Proceedings

Boston, Massachusetts
July 14–17, 1997

Sponsored by
The USENIX Association

USENIX®

The Advanced Computing
Systems Association

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $22 for members and $28 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past Tcl/Tk Proceedings

4th Tcl/Tk	July 1996	Monterey, California	$22/28
3rd Tcl/Tk	July 1995	Toronto, Canada	$29/34

ISBN 1-880446-87-1

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Fifth Annual Tcl/Tk Workshop

July 14-17, 1997
Boston, Massachusetts

Workshop Organizers

Program Co-Chairs
Joseph A. Konstan, University of Minnesota
Brent Welch, Sun Microsystems Laboratories, Inc.

Program Committee
Dave Beazley, University of Utah
Mark Harrison, DSC Communications Corporation
Jeffrey Hobbs, University of Oregon
George Howlett, Bell Labs Innovations for Lucent Technologies
Ray Johnson, Sun Microsystems Laboratories, Inc.
Kevin Kenny, General Electric Corporate R & D
Gerald Lester, Computerized Processes Unlimited, Inc.
Don Libes, NIST
John Robert LoVerso, Open Group Research Institute
Michael J. McLennan, Bell Labs Innovations for Lucent Technologies
Brian C. Smith, Cornell University

Tutorial Program Coordinator
Daniel V. Klein, USENIX

Conference Planner
Judith F. DesHarnais, USENIX

USENIX Executive Director
Ellie Young

USENIX Publications Director
Eileen Cohen

USENIX Marketing Director
Zanna Knight

Table of Contents

The Fifth Annual Tcl/Tk Workshop

July 14-17, 1997
Boston, Massachusetts

Preface

Welcome to the Fifth Annual Tcl/Tk Workshop! Like us, you are probably overwhelmed by the sheer volume of new developments surrounding Tcl and Tk. Each year, there are more system developments, applications, and extensions. We hope that the Tcl/Tk Workshop, sponsored by the USENIX Association, helps you keep current on these developments, and that the workshop proceedings serves as a valuable reference.

The workshop addresses research and development related to Tcl and Tk. Presentations range from technical details of compiler or namespace implementation to interesting and novel extensions, to experience reports from innovative and challenging applications. All published papers were reviewed by a program committee representing the wide variety of Tcl/Tk developers.

This year, in response to the high volume of quality submissions, the workshop includes a poster session where attendees can interact with the authors of more than a dozen high-quality technical posters. These posters cover the same range of material as the papers, but present content that is better communicated visually and in small groups. Poster abstracts are also included in the proceedings.

Plenary sessions include an opening keynote address by Brian Kernighan, a session with Tcl/Tk creator and champion John Ousterhout, and a panel addressing the future of Tcl/Tk. In addition to formal presentations, birds-of-a-feather sessions (BOFs), work-in-progress reports (WIPs), commercial product demonstrations, and informal demonstrations give all attendees an opportunity to share their experiences and influence the future directions of Tcl and Tk.

Whether you are an experienced Tcl/Tk developer, or are interested in evaluating Tcl/Tk for future use, we hope you find the Fifth Annual Tcl/Tk Workshop to be an educational, useful, and enjoyable event.

We hope to see you again at the Sixth Annual Tcl/Tk Workshop!

Joseph A. Konstan & Brent Welch
Program Co-Chairs, Tcl/Tk Workshop 1997

Tcl in AltaVista Forum

David Griffin
AltaVista Internet Software Inc.
dave.griffin@altavista-software.com

Abstract

AltaVista Forum™ is an award-winning collaboration environment based on the open technologies of the World-Wide Web and built on the foundation of the Tcl language [pob]. Using Tcl's inherent extensibility the AltaVista Forum toolkit provides a simple class/inheritance mechanism, an information manager customized for the data storage needs of collaboration applications, and a growing toolkit for creating asynchronous collaboration applications on the Web.

This paper details how Tcl has been employed as the basis of a commercial web-based collaboration environment. While the focus of the paper will be on the design and our use of Tcl in AltaVista Forum 98, we will also touch on how the design has evolved over the past three releases of the product.

Selecting the Tcl Language

AltaVista Forum (formerly Workgroup Web Forum) is described pretty thoroughly from a functional standpoint in [Chiu]. If all has gone well, we will have released the third version of this product since its debut in 1995: spanning the use of Tcl 7.3 through 7.6 (and looking forward to Tcl 8). (More information can be found at: http://www.altavista.software.digital.com/forum)

In late 1994 a small team of engineers in Digital's Networks group began pursuing two goals: building products which worked within the framework of the then emerging World-Wide Web and building them in a new way which matched the rapid pace of evolution expected in this space.

Rapid development, flexibility, and platform independence pointed us towards interpreted languages as a core technology for constructing these new products. Perl, Python, Tcl, and a number of other interpreters were examined and evaluated. Tcl emerged as the tool of choice and in a matter of weeks a toolbox of extensions began to come together along with a flurry of prototypes.

The selection of Tcl as a core technology came about as a compromise based on a variety of factors and, to be sure, a bit of cultural pressure within Digital. For example, Perl was arguably more popular and therefore would be more attractive to people who wanted to program in our environment. Python had a very attractive object model and reasonably clean extension mechanism.

Tcl was selected because it was a mature (version 7 and still cooking), portable (most UNIX® platforms, Windows®, and Macintosh®), and highly extensible language that possessed a vibrant user community (comp.lang.tcl), an excellent book available in most bookstores [Ousterhout], and (as luck would have it) a commitment of support by a respected engineering company (Sun Microsystems). We've been generally happy with our choice.

Initial Design Elements

Following in the footsteps of Tk (and others) we created the concept of the AltaVista Forum "Toolkit": a custom scripting language for developing a class of web-based collaborative applications. Using Tcl as the core language we extended it with both public domain and privately developed code. The goal was to develop a language that was sufficiently powerful so that simple collaborative applications could be scripted in just a few pages, but would also allow for the development of sophisticated and customizable applications as well. We also wanted to provide a language which would execute identically on multiple target platforms (specifically UNIX and Windows NT™) while not excluding the ability to use Tcl to its fullest extent (we don't disable any commands, including `exec`),

Data Management

The ndbm library from Berkeley formed the base of our database manager. We constructed a simple Tcl-object interface for it and ported it to Windows NT. (In our third release we also added a simple journalling and recovery facility to improve reliability).

Simple key/value data management was too low-level for the applications we were envisioning, so we created an "information manager" which was layered on top of the ndbm object. Initially written entirely in Tcl, this provided for hierarchically structured data access with named fields (attributes) which could be dynamically modified without management intervention (to match the dynamic nature of an interpreted environment), along with a general purpose "properties" facility. This was the basis for our "structured" data management, with the file system holding our unstructured data (the "blobs" of information that were the heart of the content we were managing).

Critical to our vision of collaborative data management was augmenting the database with a content retrieval mechanism: a search engine. We initially designed a set of Tcl commands which would allow multiple search engines to be used. For our "out of the box" product we decided to use a Digital-developed search engine called "NI2" - which would later become the core of the AltaVista Search engine. While the interface for using multiple search engines still exists, the capabilities of the NI2 facility are so heavily used that it would be difficult to integrate another search engine.

Web Tools for Web Apps

Because we were a web-based application, we needed to both interact with hypertext servers as "CGI" applications, and generate results using the HTML language. Glenn Trewitt of Digital's Network Systems Lab had released a paper on using Tcl for HTML forms processing [Trewitt] along with some C libraries. We took this code, ported it to Windows NT, and added other features that we required. In parallel, the "HTML library" took form as a collection of Tcl commands and objects which attempted to abstract the various parts of HTML page generation.

The Toolbox

Rounding out the toolkit was a suite of commands that we felt made it easier to write compact and portable scripts.

The platform command set provided for platform-independent file management (copy, delete, rename). Another set of commands managed internal date/time stamps. (Both of these facilities have been partly or completely replicated in recent releases of the Tcl core).

AltaVista Forum operates in several languages (English, French, German, Spanish, and Japanese). The toolkit provides the ability to internationalize applications using "native language tags" embedded in the code and to build catalogs that can be translated later.

A few low-level programming constructs were added as well:

- The `try-else` command is a handy abstraction for Tcl's `catch` command which has become rather popular with our developers.
- The `isnull` and `strequal` commands are shorthands for the Tcl `string compare` command. Tcl's relational expressions can be tricked into raising errors when presented with long strings of digits.
- A "mailbox" object employs a simple SMTP mail client allowing applications to generate mail messages or to read mail messages from POP servers.

Applications, Classes, and Inheritance

The design of the toolkit attempted to encompass the ideas of objects, classes, and inheritance. incr tcl was attractive as a Tcl object framework, but we were concerned that anything which required modifications to the Tcl core might put our product at risk for Tcl patches or releases – so we opted for our own, very simple, set of object-based mechanisms.

AltaVista Forum applications are built from a set of class files and module files. Module file declarations are nothing more than glorified `source` commands. Modules were added after our applications grew to be much larger than originally envisioned and needed to be broken down into more manageable chunks. Class files are similar to modules, but instead of simply sourcing them in, their inclusion in the application is via the `forum inherit` command, which applies the toolkit's inheritance semantics.

At the outermost level an AltaVista Forum application consists of a "class file": a series of `forum` commands which declare different types of objects. These class files can also inherit other class files, allowing redefinition of the objects. This mechanism allows for customization of applications without directly modifying the applications supplied in our "out of the box" product.

The forum objects consist of an amalgam of global variables, global arrays, and procedures/commands which follow an internal naming scheme, thereby creating separate namespaces which (ideally) could coexist in a single interpreter. The more practical aspect of this mechanism is that AltaVista Forum application developers could pretty much program freely in Tcl and not worry about bumping into the toolkit infrastructure which was easily manipulated through toolkit commands. For example, the command: `forum button b1 -text "b1 text" -image b1.gif -mref b1Message` ends up as entries in the class-specific array for buttons. When the button is used later in the context of a toolbar, the object name b1 is handed to a processing routine which can determine the current class name, which determines the global array name.

The following excerpts of an AltaVista Forum application demonstrate the syntax and some of the features of the language.

```
forum class paper_demo "Tcl Paper Demo"
#
# Sample of language syntax.  This
# doesn't really do anything useful.
#

#
# Bring in standard support code
#
forum inherit _stdwgw
forum inherit _acl
forum module utiities
forum module event

#
# Extended database with additional
# attributes (fields)
#
forum attribute editStatus {
   {type text} {default "open"}
}
forum attribute allowEdit {
    {type text} {default no}
}

#
# Define some HTML form composition
# elements.
#
forum textfield t.title -mapto title \
  -cols 60 -wrap virtual \
  -label "Title" -labelid l/title
forum checkbox c.allow -mapto allowEdit \
  -label "Allow edits" -labelid l/allow \
  -labelafter

#
# Sample toolbar button and toolbar
# definitions.
#
forum button tb.showDoc -text "Show" \
  -mref "showDoc %3" -senseid 3 \
  -image tb_showdoc.gif \
  -inactiveimage tb_showdoc_grey.gif

forum toolbar mainbar {
  std.home tb.showDoc
}

#
# Sample message that lists the titles
# of child documents, with hypertext
# links to those documents by clicking
# on the title.
#
forum message listSubDocs {
    # Open database.  Do access check
    # acl_deny_dialog is inherited from
    # the _acl class.
    wim open r
    if {![acl_permits]} {
       forum_exec view acl_deny_dialog
       wim close
       return
    }
    # Parent document is passed in the
    # message arguments (from URL)
    set docId [lindex $margs 1]
    begin_response text/html normal
    emitln [html head] [title_bar]
    emitln [html body] [std_header]
    emitln [toolbar mainbar $dodId]
    emitln <h1> [nlt h/lsd1 \
       "Subdocumment List for"] \
       [aval title $docId] </h1>
    set count 0
    # Enumerate children.
    foreach_entry -childof $docId d {
        incr count
        emitln [mlink [aval title] \
             showDoc $d]  "<br>"
    }
    emitln "<p>"
    emitln [format [nlt h/lsd2 \
      "Total subdocuments: %d"] $count]
    emitln [std_trailer]
}
```

Executables

The AltaVista Forum system is controlled by three executables: the dispatcher, the butler, and the background service. While the applications are shipped as files that can be modified by customers, the toolkit is a bit more protected. The hybrid C/Tcl code was merged together with a custom derivative of the Embedded-Tk [Hipp] processor which, as the name implies, allows Tcl code to be embedded into a C program framework.

The dispatcher is the web server CGI application where most of the work in AltaVista Forum is performed. Web browsers (such as Netscape Navigator™ or Microsoft Internet Explorer) contact a hypertext server, which in turn creates a process and runs the dispatcher. The dispatcher (a Tcl interpreter extended by our toolkit with a fixed initialization script) analyzes the request, loads the application, executes the appropriate scripts which ultimately generate an HTML response which is transmitted to the web browser for formatting. The dispatcher then runs down and the process is deleted.

The butler is the AltaVista Forum's version of `tclsh`. It has most of the same extensions that the dispatcher has (minus the elements only of use to a CGI program), and includes a number of forum management commands, some application development aids, and a reasonable reproduction of the dispatcher execution environment suited to offline processing.

The background service is a program that coordinates the execution of tasks that are to be performed on a particular schedule or as a reaction to some event in the applications. The background process doesn't actually do any of the work but instead exec's butlers with small scripts.

Code Volumes

The table below chronicles the growth of the toolkit proper (this does *not* count the code in various components we integrate into the toolkit such as Tcl, ndbm, NI2, etc.)

	C		Tcl	
Release	# Files	Lines	# Files	Lines
1.0	15	5,435	19	9,039
1.0A	15	7,338	19	8,900
2.0	15	9,286	19	12,994
2.0A	15	10,027	19	13,505
3.0	32	18,829	28	14,883

Compared to the applications, the use of Tcl in the toolkit has grown only modestly. The numbers reveal a bit of the toolkit team's development strategy: where possible first implement the functions needed in Tcl and then recode in C to gain performance after the interfaces and behavior have stabilized

The table below shows the growth in the forum applications code volume (essentially lines of Tcl). The ten-fold increase in size is due in part to new applications, a large number of new features being implemented, and a wider range of engineer experience with Tcl. It also reflects the toolkit lagging behind the applications (there are more application engineers than toolkit engineers) which is made up in lots of Tcl code. With a little luck, this number might actually start to decrease in future releases as we continue to fine-tune the toolkit.

.RELEASE	# FILES	LINES
1.0	8	6,700
1.0A	8	17,104
2.0	30	48,398
2.0A	34	56,773
3.0	114	97,591

Latest Design Elements

The design presented thus far served the first two major releases of the AltaVista Forum product quite well, but suffered from a performance problem that limited the scalability of the product. In our third release we were allowed to attack this problem rather aggressively. The resulting design utilized the Tcl core to an even higher degree.

Tcl Package Libraries

The first change was to replace the Embedded-Tk mechanism with a new facility called Tcl Package Libraries (TPL). All of the toolkit Tcl code is now placed in a single archive which is read in by the various executables - bringing in the particular modules they require. The TPL mechanism serves a number of purposes:

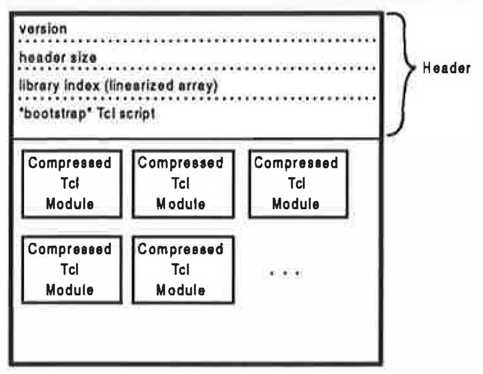
- It reduces our kit size because the Tcl code isn't replicated between 3+ programs.

- It provides a vehicle for extensive customization by resellers – an attractive feature for our product.
- It could potentially be encrypted to hide sensitive code (we don't do this in our product).
- We anticipate that Tcl source modules could be replaced by Tcl8 bytecodes [Lewis], with commensurate performance improvements by avoiding the on-the-fly compilation phase.

The TPL archive is created by a Tcl script which reads in the designated Tcl modules, compresses the comments out, and arranges them in a single file with an index structure in the header area. The archive script can also write out a "bootstrap" script into the archive which is a small script that locates and reads in the more robust package_load facility. This keeps the amount of Tcl code that needs to be embedded in the C programs to just a few lines.

TPLs are not connected in any way with "packages" as implmented in the Tcl core. The main benefit of TPLs is that all of our Tcl code in the toolkit is now bound into a single, platform-independent file which is easily distributable over the Web (for patches or incremental extensions).

The diagram below depicts the on-disk layout of the Tcl Package Library:



Structure of the AltaVista Forum Tcl Package Library

AltaVista Forum ships with a TPL containing all of the toolkit. Through configuration files the software can be instructed to load in different modules and/or additional TPLs.

Persistent Server Design

From the earliest prototypes of AltaVista Forum we knew that the high-level design of the dispatcher was nowhere near optimal for high-performance or large-scale deployment. Each transaction required a large amount of processing to occur, sometimes to do relatively little "real" work. The initial focus in release 3.0 was to quantify and then reduce this overhead before pursuing other tuning opportunities.

The first task was to instrument the dispatcher with simple probes that recorded `the [clock clicks]` at each point. A particularly slow CPU was used so that the clock resolution would not mask the results (the actual numbers used in this paper aren't nearly as important as the ratios). Here is a summarized sample of the instrumentation probes that we used as our reference transaction:

Acc. μsecs	Function
44,797	Tcl Core Initialized
52,098	AVF Toolkit (C language) Commands Installed
3,226,111	AVF Toolkit (Tcl language) Procedures Installed
4,742,309	Dispatcher Configuration and Transaction Initialization
12,370,021	Application Class Loaded
12,645,428	Transaction Preparation Complete
15,999,848	Transaction Complete

The timings show that for this 16 second transaction, about 5 seconds was engaged in fixed transaction setup, 7.6 seconds was spent loading the application into the interpreter, with the application spending less than 3.5 seconds actually doing the work (in this case the application was one of our smaller ones and the actual task was extremely simple).

This relatively simple analysis, combined with our knowledge about the growth of AltaVista Forum application sizes indicated that a major performance gain could be achieved if we reduced the time to setup for a transaction. The bulk of the application code, however, also created a constraint: whatever changes were made could not introduce the need for many modifications to the applications (the goal was zero changes required).

After prototyping our own protocol and proving the concept of a persistant server, we elected to use the FastCGI [Brown] interface as the means of interacting with the hypertext server (assuming our now familiar role of being the first to port it to Windows NT). Using FastCGI meant that we could have a process resident in memory waiting for a transaction, process it, and then set up for the next one – avoiding the fixed overhead of toolkit initialization on each CGI transaction. Referring back to our reference transaction, that gets rid of about 4.7 seconds per transaction. Because of the limited deployment of FastCGI we opted to use a CGI to FastCGI bridge program which preserved nearly all of the benefits of FastCGI while permitting us to integrate with a large range of hypertext servers.

The Interpreter Triad

A single interpreter processing these transactions was the first approach prototyped. Routines were created that would save and restore global state between transactions – however we were concerned that applications manipulating various global variables would eventually affect each other as their state was retained in memory.

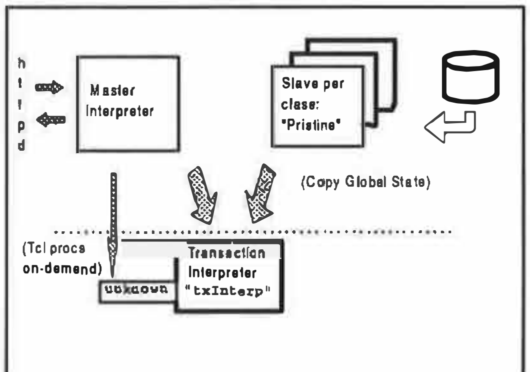
The final design incorporated the use of multiple interpreters configured in a unique way:

- The "master" interpreter holds the initialization state of the dispatcher. This interpreter accepts transactions from the hypertext server via the FastCGI interface.
- The "transaction" interpreter (`txInterp`) is a transient slave interpreter created by the master interpreter for each transaction. Ideally an application executing in this interpreter thinks it is in the same environment as the earlier releases of AltaVista Forum.
- A set of "pristine" slave interpreters are maintained by the master interpreter: one for each application class. The application is loaded into this interpreter, essentially "compiling" the application into its global state variables and procedures.

The job of `txInterp` is to perform the work of the transaction and then disappear - taking any excess global state changes with it. In the reference model it cost 8,700 microseconds for Tcl to create a slave interpreter (with teardown costing about the same) - so this overhead is pretty much lost in the noise.

The pristine interpreters exist to amortize the cost of loading the application classes (7.6 seconds in our reference transaction, and typically much, much higher) across a large number of transactions by loading it once and then keeping it in memory for the life of the process. A typical installation of AltaVista Forum has less than a dozen classes to contend with, so maintaining an interpreter per application class incurs a moderate, but reasonable memory penalty.

Commands were then written to redefine toolkit commands (C extensions) in the slave interpreters, and to quickly copy global variables from one interpreter to another. The typical cost of copying the pristine global variables to the transaction interpreter was approximately 20,000 microseconds.



Tcl Interpreters inside the AltaVista Forum "Dispatcher"

Procedure "Faulting"

Tclsh's autoloading facility provided the inspiration for the final piece of the new design. Because a transaction typically only uses a fraction of the application procedures (and the toolkit), we knew that avoiding defining all of the procedures for each transaction would save even more time. We used the `unknown` command as a "procedure faulting" mechanism and registered it as an alias in the slave interpreters. When a transaction needs a procedure that doesn't exist, the `unknown` alias intercepts it and checks the pristine and master interpreters for the procedure. If located, the procedure is transferred to the slave interpreter and executed. The original implementation of this faulter cost approximately 122 microseconds to copy a procedure, but taking advantage of access to Tcl internals this was reduced to 66 microseconds. This meant that for the first time small transactions executed much faster than the more complicated ones.

The procedure faulter is instrumented so that every procedure that flows through it is recorded in a list. We will use this information to determine the most frequently referenced commands which will become top candidates for re-coding in C.

Transaction Flow and Results

The final transaction sequence conceptually looks like this (for performance reasons some things are actually done out of order):

1. Master interpreter accepts the transaction data from the FastCGI interface and parses the incoming message storing relevant information in master interpreter global variables.
2. Create the txInterp; define the (static) toolkit commands.
3. Copy the global variables from the appropriate pristine interpreter. If there is no pristine interpreter resident for the application, then create one and load the class into it.
4. Copy the appropriate global variables from the master interpreter to the txInterp. These variables hold the per-transaction state.
5. Rig the procedure faulter to search the appropriate pristine interpreter and master interpreter for unknown commands.
6. Evaluate the appropriate transaction script in the txInterp. Send response back to the hypertext server.
7. Delete the txInterp. Go back and do it all over again.

This arrangement has proven to be extremely effective. For simple transactions a performance increase of 5X was not uncommon. Large applications developed entirely on the prior version of AltaVista Forum ran with only a handful of changes in some areas that did some operations that were probably better off in the toolkit anyway (most of the changes were related to CGI/FastCGI implementation differences and not to the interpreter design changes).

This mechanism has been essentially replicated in the butler program as well, permitting batch processing of lots of forums (application instances) relatively efficiently.

Observations

When this paper was submitted we had just completed the main development phase of AltaVista Forum 98 (Version 3), so the degree of success of the new design cannot be completely ascertained. However we have learned a few things worth noting:

By carefully isolating the developer from the platform with a wide variety of data management facilities the AltaVista Forum scripting language permitted identical execution on Windows NT, Solaris®, and Digital UNIX™.

While developers generally liked the power of the Tcl scripting language, the lack of a syntax checker and adequate debugging and performance profiling tools was often a source of distress for those accustomed to the more traditional development environments and tools. We are actively working to correct this situation.

The realities of commercial product development meant that we could not fine-tune the toolkit forever. Consequently it is not as powerful and expressive as originally hoped. Tcl's inherent power and flexibility essentially worked against us here: what the AltaVista Forum Toolkit team couldn't provide quickly enough was simply "invented around" by writing more Tcl code. This increases the size of our application code and decreases overall performance.

Our experience with Tcl continues to remain a general pleasure. It is stable and generally does what is documented. We did encounter a few problems worth noting:

- Tcl's C-level routines don't allow access to a lot of important things: like `info` command data. This meant that some our low-level code must still call `Tcl_Eval()` to execute small scripts - partially mitigating performance gains or we were forced to access Tcl internal data structures – which will make migration to future versions of Tcl more expensive.
- Tcl channel I/O and pipe interaction work just differently enough between Windows NT and UNIX to keep us on our toes. One major design feature of the background facility had to be reworked when it was discovered that it didn't work properly on Windows NT.
- We make only one modification to the Tcl core: we stub out Tcl_Init() so that slave interpreters don't need to read init.tcl.

Because a significant percentage of our product is written in Tcl, we can take advantage of this to a great extent in customer support situations. We can construct tools "on the fly" to deal with new problems, and because our configuration files are also Tcl scripts, we can "patch" misbehaving parts of the toolkit in the field without major rebuilds of the product. This feature has proved to be so valuable that we've taken pains to assure that many more parts of the toolkit are now more easily field configurable. This will increase both the supportability of the product and make it more attractive to resellers who wish to make modifications to the toolkit's behavior without necessarily changing code provided by our company.

Acknowledgments

The AltaVista Forum product is the handiwork of more talented engineers at AltaVista Internet Software than I can name here. However I'd like to particularly thank the engineers who were part of my toolkit teams located both in Littleton and Australia that developed the Version 1 and Version 2 releases and laying the foundation for the Version 3 design. Special thanks to Roy Klein, Bob Travis, and Peter Hurley who provide constant and invaluable feedback on one harebrained idea after another. Finally I'd like to thank my original partners in crime, Dah-Ming Chiu and Dave Cecil, who held the early product together, and Larry Augustus for keeping the whole boat afloat.

References

[Chiu] Chiu, Dah Ming and Griffin, David. "Building Collaboration Software for the Internet." Digital Technical Journal, Vol 8. No. 3 1996 http://www.digital.com/info/dtj

[Hipp] Hipp, Richard. "Embedded Tk" http://users.vnet.net/drh/ET.html

[Lewis] Lewis, Brian. "An On-the-fly Bytecode Compiler for Tcl", The Fourth Annual Tcl/Tk Workshop Proceedings, Monterey, California, July 10-13, 1996.

[Brown] Brown, Mark. "FastCGI: A High Performance Gateway Interface", Fifth International World Wide Web Conference, 6 May 1996, Paris France. http://www.fastcgi.com

[Ousterhout] Ousterhout, John. "Tcl and the Tk Toolkit." Addison-Wesley 1994

[pob] DataComm Magazine Hot Products Award, Jan. 1996; PC Week Lab Analyst's Choice, Feb 25, 1996; PC Magazine Editors' Choice April 1996; PC Computing 1996 Finalist MVP

[Trewitt] Trewitt, Glenn. "Using Tcl to Process HTML Forms", Unpublished Digital Network Systems Laboratory Technical Report.

"Dashboard" : A Knowledge-Based Real-Time Control Panel

De Clarke
UCO / Lick Observatory
Santa Cruz, CA 95064
de@ucolick.org

Abstract

This paper describes the use of Tcl and Tk to implement a "soft" or generic GUI for real time control systems. UCO/Lick Observatory is using Tcl/Tk in conjunction with a relational database to implement a suite of code for instrument control and observing at Keck Observatory. One Tcl/Tk application serves as both the GUI builder and the GUI. It relies on information from an authoritative database to configure its behaviour. The project illustrates the use of Tcl and Tk as the common language holding a complex project together, and the particular suitability of Tcl to database applications. It also illustrates a software design philosophy in which an online database engine is an integral part of software design and deployment, rather than the target of the application.

1 Background

1.1 How astronomers handle data

The astronomy community uses, for archival and interchange of image and tabular data, a data storage convention known as FITS. The header of a FITS file consists of a number of keyword/value pairs, embedded in a fixed record format. The FITS standard [FITS] was inspired by Hollerith card images and is somewhat restrictive. Keywords are limited to eight uppercase ASCII characters, and records are strictly constructed by column position. Here is a brief sample from a typical FITS image header.

```
SIMPLE  =                    T / FITS type
BITPIX  =                   16 / bits/pixel
NAXIS   =                    2 / image axes
NAXIS1  =                 2303 / dim in x
NAXIS2  =                 1024 / dim in y
TEMPDET =        -119.72433472 /
DEWARID =                   29 /
TEMPSET =        -119.95216370 /
DWRN2LV =          80.87911987 /
RESN2LV =          32.34042740 /
PWRBLOK =           3.17460322 /
UTBTEMP =           6.50000000 /
UTBTMPS =           5.00000000 /
UTBFANS =                    F /
AUTOSHUT=                    T /
```

A few *mandatory* FITS keywords specify the structure of the subsequent image or table data. Other, "reserved" keywords document common characteristics of data (like DATE, TELESCOPE); the rest of the keyword namespace is commonly used to supply institution-specific, application-specific, or instrument-specific information.

1.1.1 Keywords and values as a control metaphor

At Keck Observatory [KeckObs] in Hawaii, the dome, telescope, and instrument control system is based on the FITS keyword/value model, extended with numerous status and control keywords. Some keywords represent telemetry values which can be read, while others can be written to set instrument operating parameters or to move motors, close solenoids, etc. The control system metaphor is consistent with the archival data storage format, and much status information from the control system is stored in the images acquired there, along with the standard FITS keywords. The control software suite is known as KTL (Keck Task Library) and is described in Keck Software Document Number 28 [KSD28].

UCO/Lick Observatory designed and built the HIRES instrument [HIRES] at Keck-I, and is currently building the DEIMOS [DEIMOS] and ESI [ESI] instruments for use at Keck-II. During early planning for the DEIMOS instrument, the software team wanted some means of managing the large number of new (or slightly variant) keywords the instrument would need. We constructed a relational database schema for modelling FITS keywords, storing keyword attributes such as datatype, format, read/write access, semantics, etc. The schema rapidly grew in complexity, incorporating new concepts such as interkeyword syntactic and semantic relationships, internal/external representation and unit/format conversions, hierarchical keyword grouping, etc.

When nearly complete, the "keyword database" stored in this schema became a powerful resource from which we could generate documentation, sample FITS headers, and certain repetitive sections of source code; we were also able to extend the application of the schema to model database tables (groups of fields, whose attributes are nearly identical to FITS keyword attributes), and to facilitate the automatic conversion of FITS files into database records or tables and vice versa. More information about this project is available at the project Web page [Memes] and in a forthcoming ADASS paper [ADASS].

We used Tcl exclusively for the application language (basically, sophisticated report generation) and Tcl/Tk for the generic forms-based GUI to Sybase which provides our interactive access to the data. We hope to generate a significant portion of the paper documentation for the DEIMOS critical design review, as well as online documentation and substantial chunks of source code for the finished system, by means of these Tcl applications.

2 The "Dashboard" application: a soft realtime GUI

2.1 Functional requirements

Another challenge facing us was our need for a good GUI (or several) for this very complex instrument. We require an engineering interface for bench tests, development, pre-ship qualifications, etc.; and we require a finished, friendly, highly documented GUI for the end-user (astronomer) who will use the instrument at Keck-II. In the past we have hand-crafted GUIs using C and the Xt toolkit, and some personnel at CARA have used the commercial DataViews [DV] product extensively to design custom interface panels for instruments or instrument subsystems.

Our approach to the generation of documentation, code, etc., convinced us of the significant benefits of maintaining one authoritative source of keyword (i.e., design and specification) information to be used by many applications. It was my personal conviction that the UI should not be a specific, handcrafted product tailored for DEIMOS, but a generic "soft" application capable of reading the keyword database and configuring its behaviour accordingly.

The GUI should provide the user with a body of knowledge about keywords and their legitimate use, and with a "toolbox" of graphical and text widgets which could be associated with these keywords. The end result would be a control GUI for any KTL control system (really, for any keyword/value control system). It should not rely on any commercial or license-restricted software; although we started this project using the Sybase RDBMS, one could use the free PostgreSQL RDBMS [PgSQL] with the pgtcl extension). It should be portable to any Unix platform (including Linux on laptops).

In sum, the finished product should be a dashboard *builder*, not just a dashboard. The UI that ships with DEIMOS should be merely one frozen layout created by an inherently dynamic tool. If we could apply the same tool to any KTL system, then the UI for ESI and for DEIMOS would share development costs, and it should be cheap and easy to design supplemental or improved GUI for instruments already deployed. It should be absolutely trivial to change the surface appearance of the GUI at any time, in response to user requests or operational/procedural changes.

2.2 Implementation: how it works

The "Dashboard" application is a fairly slender body of code relying heavily on a large infrastructure (FIGURE 1). It requires a set of Tcl extensions as well as KTL shareable object libraries, and access to the keyword database.

Mr. William Lupton of CARA wrote the "ktcl" extension [KSD98] which adds the KTL API to Tcl. Through this API the application can open a KTL "service", register interest in automatic broadcast updates of KTL keywords, and read/write individual keyword values. "Dashboard" also relies on TclX [TclX], and can use the VU widgets [VUW], the BLT plot widget [BLT], and the LLNL TurnDial widget [TurnDial]. It is fairly trivial to add

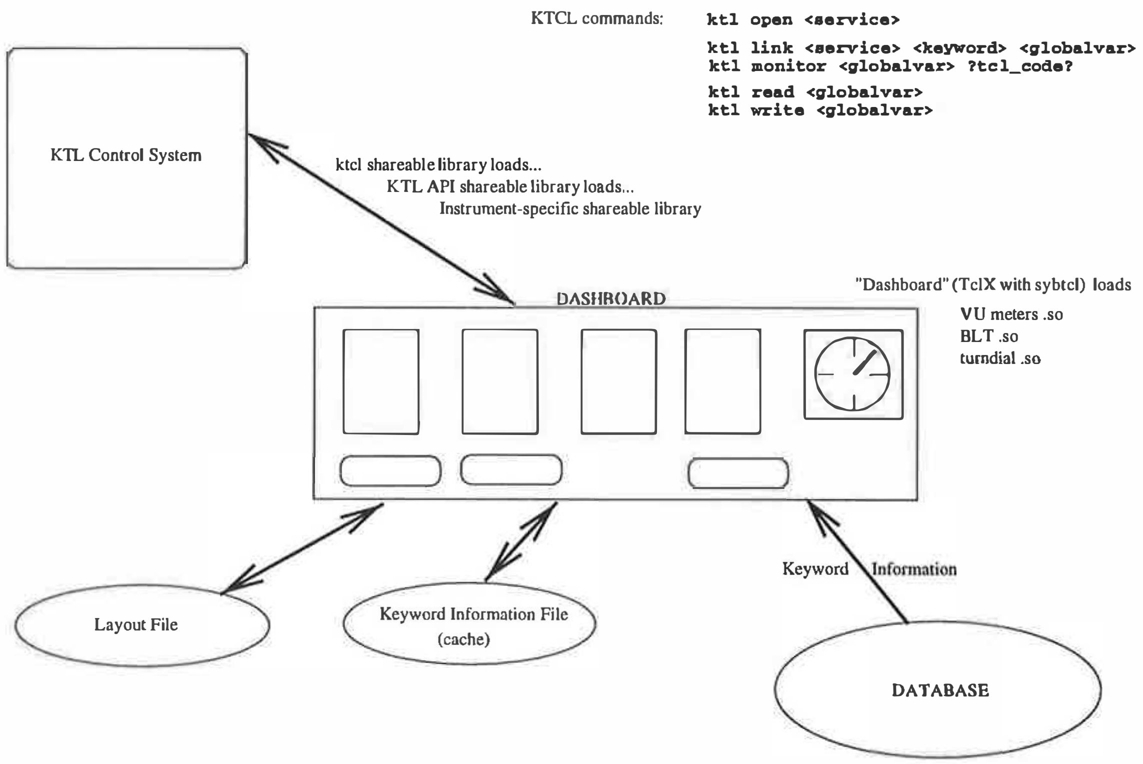

KTCL commands:
ktl open <service>
ktl link <service> <keyword> <globalvar>
ktl monitor <globalvar> ?tcl_code?
ktl read <globalvar>
ktl write <globalvar>
KTL Control System
ktcl shareable library loads...
KTL API shareable library loads...
Instrument-specific shareable library
DASHBOARD
"Dashboard" (TclX with sybtcl) loads
VU meters .so
BLT .so
turndial .so
Keyword Information
Layout File
Keyword Information File (cache)
DATABASE

Figure 1: Dashboard Function Diagram

new "meter" types to the dashboard. We used the Sybtcl [SybTcl] extension for access to the keyword database.

The application reads from the database (or from an ASCII save file of information originally read from the database) the complete descriptions of a set of keywords corresponding to a KTL "service", i.e. the keywords relevant to a particular instrument or subsystem. It creates a canvas on which the user/designer can deploy "meters" and "graphics". Meters are associated directly with keywords, and display the keyword values. Some meters can also be used to set keyword values. Graphics can decorate and annotate the dashboard, and change their state to flag various conditions (more on this later).

How is this done? The KTL API extension implements a **ktl link** command which associates any given keyword with a global Tcl variable; the global variable can then be used as the argument to a **ktl read** or **ktl monitor** command. If the keyword is being monitored, the KTL control system sends out broadcast messages on each value change, to all processes which have registered an interest in that keyword. The ktcl extension responds to these broadcasts by automatically updating the global variable with the new value (and optionally, by executing a user-defined Tcl code section). The application can explicitly read the value at any time; however, network traffic is reduced by permitting the KTL control system to broadcast values only when they change.

A detailed description of how KTL and the telescope/dome/instrument control mechanism really work is beyond the scope of this present paper. However, for those who are interested in realtime control systems, the Keck Software Documents previously cited plus a Lick Observatory Technical Report [Music] should provide an overview of the infrastructure of which "Dashboard" is the topmost layer.

When the dashboard user creates (e.g.) a "TextBox" meter using the Tk entry widget, the application simply uses the global variable created by **ktl link** as the textvariable for the entry widget, and the display is automatically updated on each KTL broadcast. For widgets which do not have the "associated textvariable" feature, the optional Tcl callback code is used to execute an update procedure for that meter type. In addition to the global

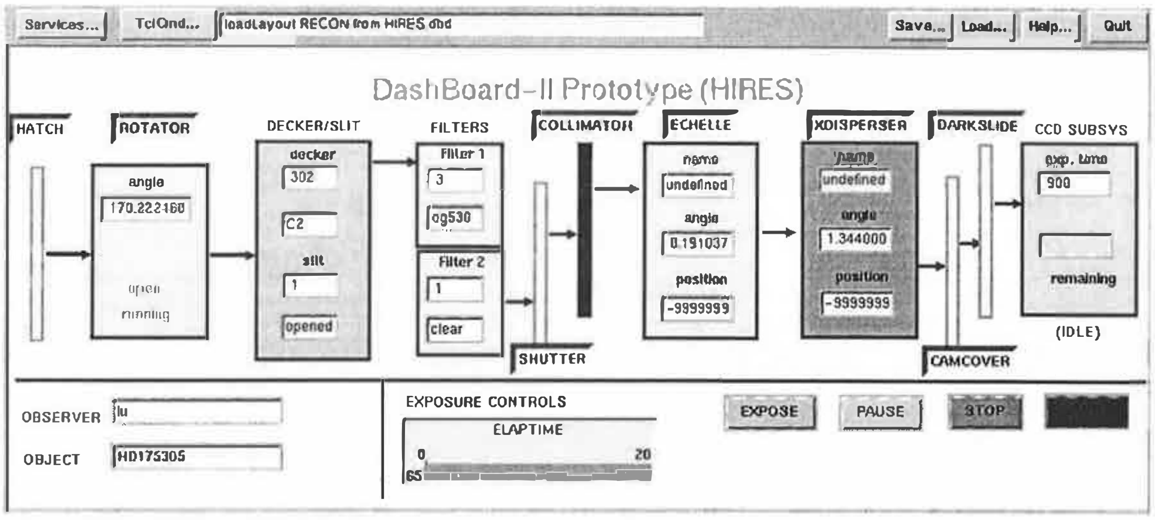


Figure 2: Prototype dashboard for HIRES instrument

variable created by `ktl link`, the application maintains for each keyword a "desired value" variable, which the user can adjust; a Commit operation is supported in which the user's "desired values" are written back to the KTL system.

In detail, the application maintains for each keyword a list of Tcl commands to be executed each time that keyword changes. The callback code uses the keyword name as an index into an array of these lists of commands, retrieves the list, then evaluates each command. Thus any number of meters dependent on one keyword can be be updated when that keyword value changes; each command is `eval`'d, so any number of meters dependent on one keyword value are all updated by the one procedure call. (In the first draft of the code, I used the Tcl `trace` mechanism; but using the callback code option of the `ktl monitor` command reduced the lines of code slightly and looked cleaner.)

2.3 Implementation: features

Using a canvas as the dashboard surface, it was easy to set up bindings for positioning meters by dragging, and bindings for editing meter configurations interactively. Only slightly more challenging was the "detachable" feature, which permits the user to detach a meter entirely from the dashboard into a separate toplevel, then replace it in its correct location at will.

For end-user convenience, the dashboard supports "pseudokeywords": the user/designer can define a pseudokeyword whose value is an evaluable expression involving numeric constants, global variables, and one or more other keyword names (thus a pseudokeyword `REMTIME` can be defined as `EXPTIME - ELAPTIME` so that the user can easily make a count-down meter measuring remaining exposure seconds). The expressions are entered in a simplified form which is expanded and sanity-checked by the application before being evaluated. Expressions are stored in both simple (for user editing) and expanded form.

Graphics are used to decorate the canvas, providing text labels, lines to delimit groups of related widgets, and geometric shapes symbolizing, e.g. hardware or software subsystems with which the user can interact. Bindings for positioning and editing graphical objects are consistent with those for meters.

The ability to evaluate expressions (for pseudokeywords) is also used to implement "conditions", boolean expressions involving keywords and global variables, whose evaluated result can be used to determine the configuration of dashboard elements. Each meter or graphical object (including the canvas itself) can be associated with not just one set of attributes, but an array of attribute sets. Each attribute set is associated with a condition controlling the application of those attributes; the base set is associated with a null or "Normal" condition. The user/designer can easily and quickly add more attribute sets and conditions, to make objects on the dashboard surface change their appearance (such as content, size, background/foreground colour, etc.) in response to KTL events. The designer/user edits meter and graphic attributes, establishes conditions, defines pseudokeywords, etc. using GUI forms invoked interactively from the dashboard. The results of these changes are immediately visible.

The application can be configured to display a "transparent" graphic for an open shutter (value of keyword SHUTTER is "open"), and a black (or larger, or both) graphic when the shutter is closed; a large red warning message can appear on the dashboard surface in response to an undesirable condition. Meters can "turn red" when the value they represent exceeds a certain limit; buttons can become inoperable when the instrument condition prohibits their associated action. The attribute set of dashboard objects is the list of their native Tk attributes, plus a small superset for the designer's convenience (`visible` is one such convenient attribute).

In Figure 2, for example, the HATCH rectangle will be solid black when the hatch is closed, but hollow (as shown) when the hatch is open. The horizontal arrows will appear when light is passing between the stages of the instrument at the points indicated, and will disappear when no light is present at those points. The Image Rotator object will turn gray and fade almost into the background when the rotator is out of the light path. Bitmaps representing glowing light bulbs will appear when comparison lamps are turned on . . . and so forth. The chain of "photon arrows" is probably the most challenging of these animation problems, and requires a chain of pseudokeywords with definitions as follows:

```
HATLIGHT
  "HATOPEN == 'open'"

ROTLIGHT
 "( ( HATLIGHT ) && ( ( IROTOUT ==
  'out of light path' ) ||
  ( IROTCVOP == 'opened' ) ) )"

LAMLIGHT
  "LAMPNAME != 'none' "

DSLIGHT
  "( ROTLIGHT || LAMLIGHT ) && ( SLITWID > 0 )"

SHUTLIGHT
  "( DSLIGHT ) && ( SHUTCLOS != 'closed' )"

COLLIGHT
  "( SHUTLIGHT ) &&
  ( ( ( COLLBLUE == 'blue' ) &&
  ( BCCVOPEN == 'opened' ) ) ||
  ( ( COLLRED == 'red' ) &&
  ( RCCVOPEN == 'opened' ) ) )"

...
```

The application can now use those pseudokeywords for conditional configuration of graphics. The arrow between the Rotator and the Decker/Slit modules has two conditions: invisible (normal) and "lit". The "lit" condition is contingent on either a lamp being on or light getting through the rotator:

```
LAMLIGHT || ROTLIGHT
```

The GUI designer needs more flexibility in responding to conditions than merely the alteration of the GUI appearance. Evaluable conditions are also used in a simple error/warning message system with popups, mailed or displayed messages, and optional Tcl code to execute on any condition. The UI designer can easily cause any arbitrary action to take place (including execs of shell commands, delivery of mail messages, etc.) upon keyword-related conditions. In this case the object is an Alarm, and has an associated array of conditions and actions much like the condition/attribute arrays for visible objects. Figure 3 shows the Alarm Editor.

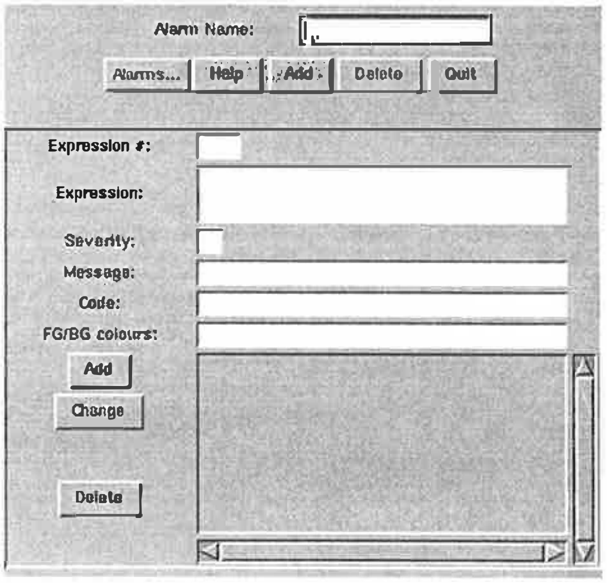


Figure 3: Pop-up editor for defining Alarms

Here is a sample alarm condition in simplified and expanded notation. The simplified notation is what the designer entered into the Condition box in the Alarm Editor.

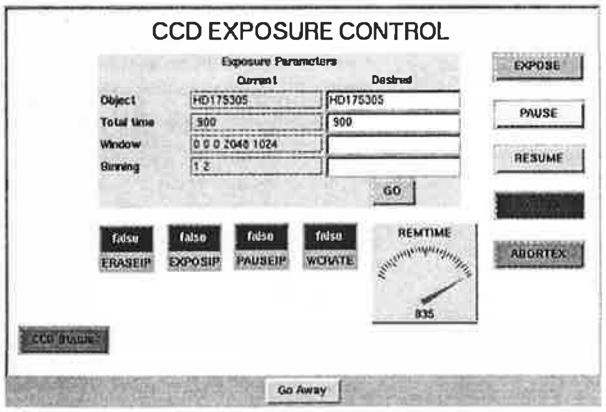
```
( LAMPNAME == 'none' ) &&
  ( LMIRRIN == 'in light path' )

( \$hires(LAMPNAME) == 'none' ) &&
  ( \$hires(LMIRRIN) == 'in light path' )
```

As you can see, the expansion process consists of converting keyword names and pseudokeyword names to Tcl global variable references. After this, the expression can be evaluated at uplevel #0 (evaluation in this case takes place on any change to the value of either `LAMPNAME` or `LMIRRIN`). If this expression is true when evaluated, a message will appear in a popup alert box:

```
No lamps, yet lamp mirror in.
Occultation?
```

Upwards of 400 keywords are needed to control and monitor the DEIMOS instrument; obviously there is not enough screen real estate on even the largest standard monitors to display this much information at any reasonable size, nor can the average user perceive and understand that much information at one time. Therefore the dashboard needs hierarchy; it can invoke and dismiss sub-dashboards offering detailed control, while presenting an overview of the system at the topmost, introductory screen. The mechanism for invoking sub-dashboards is the familiar and intuitive "double-click" used in many window systems to unfold or invoke windows, bound to any graphical object; however, the user/designer can attach any Tcl code to the double-click binding, as well as or instead of the procedure call to invoke a subdashboard.



Exposure Parameters

	Current	Desired
Object	HD175305	HD175305
Total time	900	900
Window	0 0 0 2048 1024	
Binning	1 2	

ERASEIP	EXPOSIP	PAUSEIP	WCRATE
false	false	false	false

Figure 4: Sub-dashboard invoked from HIRES main dashboard

Figure 4 shows a subdashboard which was invoked from the CCD stage of the main HIRES dashboard with a double-click. This subdashboard offers more detailed control of the CCD control subsystem. The "CCD Status" button offers a third level of detail (Figure 5) in which a BLT graph widget is used to plot detector temperature against the liquid nitrogen level in the dewar. The PS button makes a PostScript file of the current plot.

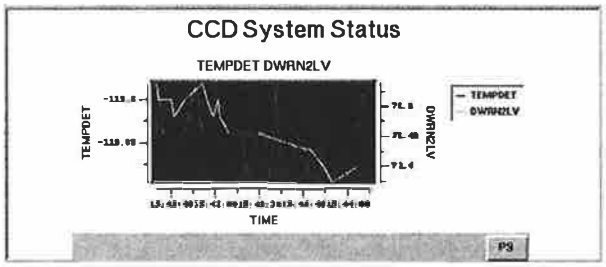


Figure 5: Sub-sub-dashboard invoked from CCD Status button

For design flexibility, the dashboard supports three hybrid screen items: a generic Button, generic Menu, and generic Entry widget. Unlike a Meter, these items are not bound to any particular keyword, and unlike a Graphic they are standard Tk widgets embedded in a canvas, not native canvas items. The user/designer can configure the generic menubutton and button's "command" attribute freely, to execute any arbitrary Tcl code; the generic entry widget can be associated with any global variable (for example, if the application requires the UI to display Desired as well as Actual values simultaneously). These generic widgets support the same conditional attribute mechanism as the other dashboard components. (Example: several of the stage labels in Figure 1 are actually menus offering control of the covers or other basic functions for the optical stages. The menu items would be, e.g., "Open Cover," "Close Cover," "Into Light Path," "Out of Light Path.")

The dashboard operates in several (nonexclusive) modes. In "engineering" mode, the user can edit the dashboard freely; each writable Meter has individual edit and commit buttons, and some extra function buttons are provided in the frames around the central canvas. In "observer" mode, the screen is smaller and less complicated, omitting extra function buttons and invidual Meter buttons. In "developer" mode, Meters have large obtrusive frames which make them easy to reposition and edit, whereas in "deployment" mode these frames are invisible and the user can neither reposition nor edit items on the screen. In "safety" mode, where any KTL write command would normally have been executed the application will instead log a message to stderr. In "fake" mode, the dashboard will not even try to connect to a KTL service but will use fake values for all keywords; this permits early design to be done before the supporting keyword libraries are compiled, or before a running instrument control system is available.

Release 1 of the dashboard knows how to use the VU Dial Gauge, Bar Gauge, and Stripchart meters, the BLT plot meter, and the TkTurndial meter. It also provides a TextBox (decorated entry widget), Odometer (undecorated entry widget), Grid (gridded frame of entry widgets with actual and desired values), On/Off Light, and Keyword Action button. It offers the Line, Rectangle, Oval, Arc, Polygon, and Bitmap canvas items as graphics, and the non-keyword Button, Menu, and Entry. It supports conditional configuration of all meters and graphics, and delivery of alarms (or any arbitrary Tcl code execution) on any condition. A command-line window and a simple script editor are provided for the user, for the direct typing of Tcl commands as an alternative to GUI interaction. Screen layout can be saved to and reloaded from plain ASCII files; all pseudokeyword and condition information is saved as well as the layout and configuration of the main and all sub-dashboards.

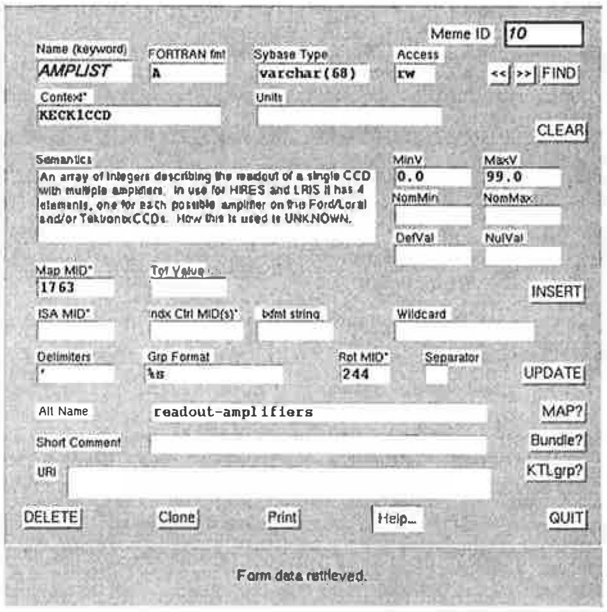
3 Positive aspects of the "Dashboard" approach

3.1 Practical advantages

The immediate appeal of the dashboard is in its flexibility and the speed with which interfaces to KTL-based instruments can be generated. The interface shown in Figure 2 was created just as a demo, to exercise and debug the first release of "Dashboard"; it took about 4 hours to create the toplevel screen, and about another 3 hours to create four subdashboards (which pop up on double-clicks from the main board).

The dashboard is also (usually) "live" during design and prototyping; there is no compile/link/test cycle. As soon as the keyword information is absorbed and a widget is created, that widget is "real", watching a real keyword value in a running KTL system (unless the designer is running in fake mode).

If the engineers change their minds about hardware/software design or function, those changes are first reflected in the keyword database. The application then effectively retools itself to match these design changes, always provided that the keyword database is properly maintained. Since the keyword database is the foundation of a number of applications, not just one, the likelihood of its proper maintenance (and of someone's noticing any errors or inconsistencies) increases with the number of applications that depend on it. Also, the availability of interactive X11 forms (Figure 6) and other GUI and command-line tools for editing database information makes it relatively easy for developers to keep the central knowledge base current and accurate (as opposed to the manual editing of many distinct copies of the same information in the form of code, config files, man pages, typeset documents, etc.)



Meme ID	10
Name (keyword)	AMPLIST
FORTRAN fmt	A
Sybase Type	varchar(68)
Access	rw
Context*	KECK1CCD
Units	
Semantics	An array of integers describing the readout of a single CCD with multiple amplifiers. In use for HIRES and LRIS it has 4 elements, one for each possible amplifier on the Ford/Loral and/or TektronixCCDs. How this is used is UNKNOWN.
MinV	0.0
MaxV	99.0
NomMin	
NomMax	
DefVal	
NulVal	
Map MID*	1763
Tol Value	
ISA MID*	
ndx Ctrl MID(s)*	
bfmt string	
Wildcard	
Delimiters	'
Grp Format	%s
Rot MID*	244
Separator	
All Name	readout-amplifiers
Short Comment	
URl	

Form data retrieved.

Figure 6: Typical GUI form used to edit keyword data

The dashboard is a single application, with one maintenance cycle and one investment in development, which can be used to provide engineering/diagnostic interfaces, deployed user interfaces, prototype UI designs, etc., for any number of instruments sharing the KTL control protocol. In the past, we developed individual interfaces per instrument per application, and these interfaces were burdensome to improve or repair; as a result, user complaints and feature requests were seldom resolved. The "softness" of the dashboard means that the advanced user can tweak its appearance to suit his/her own tastes, or design entirely original personal dashboards for specific purposes; the deployment mode means that it can be given to naive end users as a "canned" UI. The designer who must support a deployed version can easily and quickly implement most user requests, without tedious recoding and recompilation, by replacing a single platform-independent layout file.

Being based entirely on publicly-available code, the dashboard is free and portable. It runs as well on our Linux laptop, or my Linux home computer, as it does on Dec Alpha or Sun Sparc platforms.

3.2 Design philosophy

All of the above features result in cost and time savings. However, most of them are merely side-effects of what is in my opinion the single really interesting feature of the dashboard, which is its symbiotic relationship to a knowledge base embodied in an online database. In most Tcl/Tk applications involving databases, the database is the *target* of the application; i.e., the information in the database is manipulated by the application. In my "fosql" Tk forms GUI for Sybase, the forms designs themselves were stored in Sybase tables, and the data in the primary forms design table could be edited using the form for that table. However, the *target* of the fosql application was still the data in the rest of the database.

In the "Dashboard" application, the database is not the target of the app; the telescope, dome, and instrument and the data gathered by the instrument are the target of the app. The database is an intrinsic part of the application itself. It contains "information about information", or "meta-data" which the application uses to configure itself and to make certain inferences about its own function. This could be seen as one way of implementing object orientation; a database is the ideal way of representing objects and their attributes, and the code is merely the methods which apply to the objects. We could also regard the database as the equivalent of hundreds or thousands of C source "structs".

Many other applications, such as mail tools, use resource files of one kind or another to configure themselves, thus avoiding recompilation across changes of appearance or function; but in general, each application has its own resource file which is not shared with any other applications. (The X resource database is one exception, being a true online database with a known API, but most X clients have private resource files applicable only to their instance or their class.)

In contrast to the "private resource file" model, a central authoritative body of knowledge about the information on which the application operates – implemented as an online database – forms a conceptual hub about which many applications (such as the dashboard) can be constructed with maximum generality at relatively low cost. The tedious and repetitive type of tabular information which (in our older control systems) is replicated many times in different C sources, vxWorks sources, etc. is here available in one consistent place, accessible online. (It can be cached as FITS files or other ASCII formats in case we lose access temporarily to the live source.)

I should perhaps note here that as well as keyword syntax and semantics, overall system design is also expressed in our database as tables of hardware and software "agents," which pass "keyword" information between them in various formats and media. Using the digraph tools from Lucent Technologies [Dot] we can easily generate information flowcharts for the hardware and software subsystems. Thus the majority of our project design information is online in a highly standardized, codified, machine-readable form; this in turn means that 80 percent or more of our project documents are auto-generable.

The logical conclusion of this conceptual strategy is that design, documentation, and by inference a certain percentage of generable source code, are all manifestations of one body of knowledge, expressed once and maintainable at one central point.

All the usual reasons for choosing Tcl/Tk apply: speed of prototyping and development, low cost of modification and deployment, portability, lack of commercial restraint on distribution. However, there were certain project-specific reasons as well. From prior experience I had already become convinced that Tcl(X) was a near-ideal language for database applications (largely because of its typelessness and its solid list processing features). Lastly, because of the `eval` feature combined with the above, it is remarkably easy to write self-

configuring multi-purpose applications in Tcl (using methods which have no equivalent in C or other compiled languages, such as the dynamic generation of variable names and on-the-fly generation and execution of code). The "Dashboard" application was a logical outcome of previous positive experience with Tcl and its extensions.
So far, we feel that the "Dashboard" application has been a Tcl success story.

4 Potential applications

Nothing restricts the use of the "Dashboard" application to astronomy or to KTL systems. Any keyword/control system with an API could use the dashboard code, by writing a different Tcl extension and altering about 30 of the 12K lines of code in the dashboard-II application. The "trace" mechanism could substitute for the monitor/callback mechanism, or polling could be implemented for a control system with no monitor/broadcast facility. Dashboard is a shell, in other words, which could be inhabited by applications other than KTL, just as the "FITS keyword" database is a shell which could be inhabited by other kinds of syntactic and semantic information.

5 Future plans

Since the dashboard code is basically object-oriented, one might ask why I didn't use extended [incr Tcl] instead of plain TclX to implement it. An object-oriented Tcl would have been a more natural choice; I implemented some OO-like features the hard way. When itcl becomes a standard extension requiring no core modifications, I'll probably convert "Dashboard" to itcl, and the code will then become smaller and cleaner.

The code currently saves its layout to an ASCII file (Tcl source). I would prefer that it saved this information to a database schema, like its ancestor the "fosql" package. However, development was so rapid during the first six months that I decided to skip the overhead of schema revision and work from flat files. Storing the layouts in database tables would offer far more power and ease of access for global modifications, queries about the tool design, hunting down particular widgets and bindings, etc. I am used to this convenience in the "fosql" package and am already feeling the lack of it.

There is currently no provision for communication between multiple running copies of the dashboard. This seems a serious shortcoming, one which must be remedied before ship date. Since partnered observing is very common (remote observer on the mainland or in Waimea, in close communication with observers on the summit), the ability to share dashboard configurations and information seems essential. This raises all the usual issues of distributed applications (registry, trust, etc) and rather than reinventing all those wheels I'll probably evaluate existing Tcl distributed applictions and copy or adapt a successful design (with the author's permission).

The code does not yet use KTL features like "callback on move complete" which would permit us visual indication of moves in progress, and also to detect and visually flag any drift in stage positions. We are still struggling (on the KTL side) with issues like "estimated time to completion," "stage position tolerance," and so forth. However, once we have decided how to encode these concepts in the schema, or how to implement them as keywords with readable values, it will be trivial to "teach" the dashboard how to use them to construct conditional behaviours.

6 Acknowledgments

The application would not be half as successful without a few very useful features of Tcl/Tk and the TclX extension. In particular, the grid geometry manager is a very welcome new Tk feature which significantly reduced coding time; arrays of keyed lists (tclX) are a perfect analogue to database tables and hence are heavily used in most of my code. Tcl and TclX list functions, and expression evaluation, were essential. TclX's package/library system has been very useful and appropriate for code maintenance.

Much thanks to Mark Diekhans and Karl Lehenbauer (TclX), W. F. Lupton (ktcl), T.Poindexter (sybtcl), G. Howlett (BLT), P.-L. Bossart (tkTurnDial), and the original authors of the VU widget set (F. Gardner, L. Miller, R. Dearden of VU Wellington, NZ). Thanks also to Steve Allen (FITS expertise and KTL code porting) and the rest of the DEIMOS team for their ongoing suggestions and

challenges. And as always, thanks to Per Cederqvist and friends for CVS!

7 Availability

The dashboard code is freely available. Contact me if you would like to experiment with it.

References

[ADASS] De Clarke and S. L. Allen, *Practical Applications of a Relational Database of FITS Keywords*, ADASS Conference 1996 (Virginia)

[BLT] The BLT extension `ftp://ftp.neosoft.com/tcl/ftparchive/sorted/devel/BLT2.1.tar.gz`

[DEIMOS] The DEIMOS Instrument Project, `http://www.ucolick.org/~deimos`

[Dot] The graphviz package and tcldg extension `ftp://research.att.com/dist/drawdag`

[DV] The DataViews Toolkit, `http://www.dvcorp.com`

[ESI] The ESI Instrument Project, `http://www.ucolick.org/~loen/ESI/esi.html`

[FITS] The FITS Standard, `http://www.gsfc.nasa.gov/astro/fits/fits_home.html`

[HIRES] The HIRES Instrument Project, `http://www.ucolick.org/~hires`

[KeckObs] Keck Observatory, `http://www.keck.hawaii.edu:8080`

[KSD28] W. F. Lupton, KTL Programming Manual (KSD 28) `http://www.ucolick.org/~de/KSD/ksd28.ps`

[KSD98] W. F. Lupton, Tcl/Tk/KTL Interface (KSD 98) `http://www.ucolick.org/~de/KSD/ksd98.ps`

[Music] Lick Observatory Technical Reports: Music `http://www.ucolick.org/~de/KSD/music.ps`

[Memes] A Database Schema for Representing Meaning, `http://www.ucolick.org/~de/deimos/Memes`

[PgSQL] PostgreSQL, `http://www.postgresql.org`

[SybTcl] The Sybtcl extension `ftp://ftp.neosoft.com/tcl/ftparchive/sorted/databases/sybtcl-2.4`

[TclX] The tclX extension `ftp://ftp.neosoft.com/pub/tcl/TclX`

[TurnDial] The tkTurndial widget extension `ftp://redhook.llnl.gov/pub/visu/tkTurndial-2.0b.tar.gz`

[VUW] The Victoria University widgets extension `ftp://ftp.ucolick.org/pub/UCODB/VUtk41.tar.gz`

Caubweb: Detaching the Web with Tcl

John R. Lo Verso and Murray S. Mazer

The Open Group Research Institute
Eleven Cambridge Center, Cambridge MA 02142 USA
{j.loverso,m.mazer}@opengroup.org

Abstract

Caubweb^{TM} is a research system that allows a user to create local collections of Web documents on the user's computer, for access to those collections when disconnected. The system is part of a project investigating ways to provide adaptive, ongoing read and update interaction with Web-based information, even under conditions of variable or intermittent network connectivity. Caubweb is architecturally an HTTP proxy augmented with value-adding capabilities. To accommodate our design principles of platform-portability and extensibility, we used Tcl as our implementation language. This paper reports on our experience in using Tcl/Tk to build Caubweb. We discuss the structure of our implementation, identify strengths and weaknesses of the language and its tools, contrast Tcl/Tk with alternatives, and present a "call to arms" for the Tcl/Tk community, to promote increased reuse and cooperation.

Keywords: World Wide Web, Detachable Webs, HTTP proxy servers, Tcl, Tk, Disconnected operation, Mobility.

1. Introduction

Caubweb^{TM} is a research system for investigating ways to provide adaptive, ongoing read and update interaction with Web-based information, even under conditions of variable or intermittent network connectivity. Caubweb is part of our group's Distributed Clients project [17], which has the broad goal of increasing the availability and customization of Web-based information services for mobile computing users. The expected benefits include increasing the availability of information, reducing the latency of servicing requests, and adapting information to the specific user and context.

Caubweb currently focuses on support for disconnected operation, implementing caching of user-specified "weblets" for access when disconnected. Caubweb also demonstrates the staging of user changes to stored documents when disconnected and integration of those changes into origin servers upon reconnection. These features are analogous to support for disconnected operation in file systems. In addition, a prototype visualizer named CaubView (which uses library components from Caubweb) provides views of the relationships among elements in a Caubweb cache.

Caubweb is architecturally an HTTP proxy augmented with appropriate value-adding capabilities. This accommodates our design principle of *vendor neutrality*, meaning that the functionality is not restricted for use with one vendor's browser or server. This is increasingly important as Microsoft, Netscape, and others work on increasingly incompatible technology. As identified by Brooks et al. [2], vendor neutrality can be achieved in many cases by adding application-specific capabilities to HTTP proxies, which can transparently filter, transform, and otherwise process the stream of HTTP requests and responses generated by the user's browser and the Web's origin servers[22].

To accommodate our design principles of *platform-portability* and *extensibility*, we chose Tcl[19] as our implementation language. This paper reports on our experience in using Tcl/Tk to build Caubweb. We discuss the structure of our implementation, identify strengths and weaknesses of the language and its tools, contrast Tcl/Tk with alternatives, and present a "call to arms" for the Tcl/Tk community, to promote increased reuse and cooperation.

Section 2 motivates the need for information access under variable connectivity conditions. Section 3 discusses the decision to use Tcl. Section 4 discusses the structure of the Caubweb application and its substantial use of library components. Section 5 presents an evaluation of Tcl and its toolset as we experienced it in building Caubweb. Section 6 revisits the decision to use Tcl. Section 7 summarizes our findings and urges greater cooperation and interaction within the Tcl community, in order to promote increased use of Tcl as "programming for the Internet" takes on greater urgency.

2. The Detachable Web

The World Wide Web has revolutionized information access, dramatically broadening the set of users and tasks for which network-based publishing and information access has become commonplace. We now think nothing of clicking on a hyperlink that points halfway around the world to retrieve even trivial bits of information.

Traditionally, users have been constrained to accessing information resources while in designated workspaces, such as offices or homes. They are increasingly able to access these resources elsewhere, with the increased availability of portable computers and wireless and remote communication systems[13]. Nonetheless, there will be many times when the user cannot contact remote information resources, and the user must make do with the data available from the local machine (in a mode called *decoupled computing*: the ability to compute when detached from the existing computing and communications infrastructure[11]). In this narrower context, the goal of the work reported here is to provide the user of a portable computer with ongoing interaction with Web-based resources when disconnected from the network infrastructure. We call this a *Detachable Web*. Our approach applies equally well to portable and "non-portable" user machines; the key technical challenge is to cope with periods of disconnection.

The intersection of these two trends (hyperlinked multimedia documents and portable computers) offers new problems not found in either setting. For example, work in disconnected file systems did not consider support "above" the file system level and did not consider embedded object references and access to associated services (e.g., an annotation service). Likewise, the Web community has only partially examined issues such as variable bandwidth fetching and presentation, providing Web-based services while disconnected, and merging documents created or modified off-line with available on-line servers.

2.1 The Approach Taken

The essence of the approach described here is: allow the user to specify *weblets* (connected subsets of Web content) of interest; cache those weblets locally; and, when disconnected, impersonate the servers on which the cached information resides. (This last aspect is achieved by trapping the requests for those URLs and serving them out of the cache.) In addition, if the user changes a locally stored document and publishes it toward its origin server, maintain the new and previous versions while disconnected; upon reconnection, integrate the new versions back to supportive origin servers.

The system described here does not depend upon changes in or specializations to Web browsers or servers.† Caubweb, as a proxy, is placed in the middle between the browser and the server. Therefore it can catch and divert user requests appropriately, acting for the most part transparently to the user, browser, and servers. Our specialized proxy provides caching, change staging and integration, weblet specification and retrieval, presentation, proxy configuration, and control.

The proxy approach has independently been pursued by commercial "off-line browsing" software vendors (e.g., WebEx[28]), who focus on read-only access and Windows-based platforms. Browser vendors are integrating off-line browsing support more tightly with their products, and other approaches use independent applications that take advantage of browser APIs for tracking activity and request display (e.g., WebWhacker[6])

Detachable Web support is analogous to support for disconnected file access. The primary target platform is portable client machines which experience alternating intervals of connectivity and disconnectivity, both voluntary and involuntary. The minimal goal of the system is to permit read-only access to a detached web. A next step is to support modifications to the detached web and to integrate those changes into the appropriate Web servers upon reconnection. As in some disconnected file systems[14], we assume we cannot make changes to the implementation of servers (but can use existing interfaces).

The primary differences between a system for Detachable Webs and a system for disconnected file access relate to ways of discovering object references to be cached, coherence requirements, underlying object model, and the semantics of object collections. For example, systems providing disconnected file access do not make caching decisions based on references to other files contained in already cached files or external services. A key notion in a Detachable Web system is following embedded hyperlinks, images or objects in a web page being cached or viewed. Other relevant object references may come from external services (such as PICS servers)[16].

†. Hence the acronym **Caubweb: Caubweb** *A*ugments *U*ser *B*ehavior *W*ith *E*very *B*rowser

2.2 Motivation for a Proxy-based Approach

One goal was to build a system that could work with any existing off-the-shelf browser or HTTP server. This allows us to avoid the problems that arise from modifying a browser, even if we could do so. Modifying a browser limits the new functionality to be available in a specific browser version or forces the developers to race to keep up with the ever-burgeoning number of proprietary browsers and servers. Further, it is no longer reasonable to assume that one can modify the browser the user wants to use. For similar reasons, we did not want to modify any features of the underlying operating system or network framework.

The kind of support Caubweb needs to provide is best represented as middleware. Consequently, we chose to base our system on the notion of *application-specific stream transducers*[2]. A stream transducer performs some specific value-added function for the user, usually transparently. As a stream transducer between the browsers and the servers, Caubweb can catch and divert user requests appropriately, allowing us to retain the use of unmodified HTTP for a communication medium between our component and other components.

If the world were not full of huge, monolithic clients with many features hardcoded into the application, then there would be more ideal ways to add Detachable Web support. In particular, if the user-side caching model of a browser were implemented against a simple API in such a way that it was replacable when the browser was deployed, then the caching engine for our system could be used to replace it. Such a modular approach does exist for some browsers (such as Internet Explorer), but not for the bulk of the off-the-shelf browsers with which we wish to interoperate.

3. The Decision to Use Tcl

Three principles guided our choice of implementation language: platform-portability, extensibility, and ease of distribution. Platform-portability means that the system should be usable on multiple platforms with as much code re-use across platforms as possible (or, equivalently, as little platform-specialized code as possible). Extensibility means that one can extend the capabilities and functionality of the software easily, through modular software interfaces, without appealing to vendors, and without being forced to work within the constraints of restrictive licensing. Ease of distribution implies preparing as few executables as possible to accommodate the set of target platforms.

These criteria pointed toward an interpreted language, eliminating languages such as C, C++, and Visual Basic. An interpreted language allows the same code to work on different operating systems and machine architectures without any changes. Machine dependencies are hidden inside of the interpreter, yielding a high degree of portability. Interpreted languages typically provide powerful string processing capabilities, appropriate for dealing with HTML, and thereby avoiding issues of dynamic string allocation, growth, and reference management. Interpreted languages often promote rapid development (typically trading off application performance) [25]. The end-user need not compile the sources or download a platform-specialized distribution in order to use an application. Finally, interpreted languages often have graphical user interface modules that are portable across platforms, relieving the programmer of the details of different windowing systems.

The primary candidates were Java[7], Tcl, and Perl[29]. At the time of our evaluation, Java was a freshly released language with rapidly evolving language definition, program development support, runtime support, and portability. That instability recommended against Java. Perl has been used successfully by many projects, including some of our own [18]. Perl supports a reasonable object-oriented programming style, has an interpreter augmented with a byte-code compiler (improving performance), has a well-organized, cohesive, user-contributed library, and is easy to embed in other programs. Nonetheless, Perl was not generally available at that time on Windows and Macintosh platforms and lacked strong visual interface support.

These factors and our own strong experiences led us to select Tcl, which was an excellent choice for an initial implementation of the system. The primary reasons were its ease of use, its interpreted nature, its relative maturity, and, finally, its promised portability to all the major computing systems we targeted. Because of our previous experience with Tcl, we believed that the language is easy to learn and understand (permitting new team members to become productive quickly), generally allows for the creation of highly readable code (promoting reuse and transfer of code responsibility), and is immensely fun to use. Section 5 discusses in detail our perception of the strengths and weaknesses of Tcl, based on the implementation of Caubweb.

4. The Structure of Caubweb

This section describes both the Caubweb application and the library components that provide much of the core functionality. The library, *Cobweb*†, is intended to be general purpose and can support applications other than Caubweb. We first describe Caubweb's major pieces,

†. Acronym available upon request.

followed by descriptions of modules in Cobweb used to implement interesting features of Caubweb.

4.1 Caubweb

Caubweb is an implementation of a Detachable Web proxy. It typically runs as a stand-alone Tcl program, started separately from (and usually before) the user's browser. It listens and responds to HTTP proxy requests at a given port (usually 8088 with connections restricted to the local host†).

4.1.1 Usage Model

The intended usage model of the system is simple. The user first starts a personal copy of Caubweb on a laptop (or other computer); the user then configures a web browser to proxy through Caubweb. When the system is well-connected to the rest of the world (while at work, for instance), the user browses the web as normal. Caubweb, as a transparent "middleman" in the browsing activity, follows the user's actions and caches the results of the user's interactions. The user can direct Caubweb to apply a *weblet retrieval* to pre-fetch resources asynchronously, so that (to some depth) resources connected by embedded or related hyperlinks will also be available to the user later. Weblet retrieval can be explicit (the user provides both the starting URL and the retrieval criteria) or implicit (the user sets default retrieval criteria, which are applied to each URL the user requests). At some point, the user will shut the system down.

Later on, when the system is no longer connected to the Internet at large (on the airplane, for instance), the user can turn on the laptop and start Caubweb, indicating not to use the network. Caubweb will serve HTTP requests with resources available in its cache. It can also note when the user browses outside the range of the resources in the cache; these cache misses can then be used to start a new weblet retrieval the next time Caubweb is told the network is available.

The following subsections list Caubweb's major functional components.

4.1.2 Caching HTTP Proxy

Almost everything that Caubweb does (or can do) is enacted by HTTP flowing into or out of the system. As a caching HTTP proxy, Caubweb listens for incoming user requests and directs outgoing HTTP requests. However, there is not a one-to-one correspondence between the incoming and outgoing requests. Weblet pre-fetching results in Caubweb initiating its own outgoing HTTP requests. Some incoming user requests may be to Caubweb's control panel, which is original content in a portion of the URL space for which Caubweb acts purely as an HTTP server.

Almost every resource that Caubweb receives is saved in an extended HTTP cache. This cache will be the sole source of documents when Caubweb is disconnected from the network. The cache nominally follows the requirements for an HTTP cache as defined in the HTTP protocol, RFC 2068 [4]. To provide better off-line browsing ability, Caubweb caches *all* documents that pass through it, including those that a normal caching proxy would not be allowed to cache. The external behavior apparent by observing Caubweb when connected to the network will always be that of a compliant HTTP caching proxy, meaning that it will appropriately discard cached documents while connected. This behavior allows Caubweb to serve possibly stale documents when disconnected. This is likely to be exactly what the user wants, since stale data is better than no data (as long as the user is provided with appropriate cues about the data's freshness).

4.1.3 User Interface

Caubweb's normal interface to allow the user to interact with and control the system is a *control panel* provided by a set of dynamically created HTML pages. These pages allow the user to view the overall status of the system, change various preferences and settings, and get listings of (and control over) the cache contents. The various changeable controls are implemented via HTML forms, with a hyperlinked help system for cues on use of individual controls. The control panel is accessed at a special URL, `http://caubweb/`, which uses a fictitious host served internally by Caubweb.

Caubweb provides an additional, optional user interface component that is implemented using Tk. This interface provides a hierarchical status display showing the state of events occurring inside the system. The user can gain, at a quick glance, information about Caubweb's activity at any time. The status display also includes some simple controls that complement those available on the control panel. More sophisticated user interfaces are certainly possible, but this was not the focus of our work.

The HTML control panel is the primary interface, and the Tk display is optional for two reasons. First, Tk is not required for the core functionality. Caubweb works as well using tclsh as it does using wish. Requiring the use of Tk to invoke control features would mean that Caubweb could not act transparently in the background, without taking up valuable screen real estate (as is usually the problem with many Windows programs). If

†. Except on systems lacking the ability (MacOS).

Tk is not required, then Caubweb can even operate without access to a display.

Second, while the overhead of the status display is not large when using Tk with the X Window System, it is about twice as compute intensive under Windows and on MacOS. Because of this, the status display can be disabled by an option or user preference, and the system simply ignores all the status panel code when Tk is not available.

4.1.4 Weblets

A weblet is a logical collection of documents that fit user-defined criteria. Individual documents may be part of many weblets at the same time, and weblets may overlap in content. Weblet members are identified dynamically as needed. As a result, at any instant in time, it may be impossible to answer "what resources are in this weblet" because all the possible members are not known. Weblet membership is ephemeral - Caubweb does not currently keep persistent membership information.

A weblet is normally identified by some starting point (typically a URL) from which all other documents in the weblet are related. The relationship to the starting point is established by a *weblet template* which is a collection of *qualification predicates* a document needs to meet. A weblet template is applied first to the starting point of the weblet, and then, recursively, to any additional documents matched by the template. Each qualified document is parsed into a list of its internal hyperlink references, such as anchors, images, frames, etc. A document is added to the cache by virtue of this weblet if it is (1) referenced in a document already in the weblet and (2) matches the qualification predicate.

The *qualification predicate* is one of several forms of test that may be applied to a document or its properties, such as the URL. These tests include:

- a pattern match on the URL (e.g., http://www.opengroup.org/RI/*)
- a pattern match on the anchor associated with the hyperlink (e.g., "Research Institute*")
- a depth limit (e.g., no more than three links away from the start URL)
- tests against other properties (e.g., size, type, etc.)

Predicates are conjoined and disjoined to form the expression in the weblet template.

The Caubweb control panel has several convenient ways for the user to create new weblets. The simplest is a small form that handles most of the typical cases via a set of pre-defined templates. A more complicated form allows finer control, while an expert form allows arbitrary weblet templates to be specified in the weblet language.

The user can apply a template to a start URL to start a *weblet retrieval*. This causes Caubweb to start a background task fetching the documents that comprise the weblet. This allows Caubweb to cache documents that the user may never have visited.

Weblet retrievals may also be started automatically via a default weblet template. When this is enabled, Caubweb will apply the template to every URL the user's browser requests. When the user utilizes this mechanism for normal browsing, the result is a cache that is "rich around the edges" with content (possibly) related to something the user was interested in.

4.2 The Cobweb Library

As mentioned earlier, our library is intended to be highly reusable and serve multiple purposes. Cobweb contains several major subsystems, including:

- an asynchronous execution model based upon events and callbacks
- an object-oriented execution environment, based on `obTcl` [5]
- several Web-centric modules (URL, HTML, HTTP, caching, weblet, etc.)

In addition, there is a collection of other useful modules (e.g., host name resolver, splash screen, and *comm* facility for implementing Tk send(n) over sockets).

Cobweb is intended to be architecture-neutral. However, there are several places where it needed to be cognizant of the host system, usually to work around limitations or bugs on the Mac or Windows ports (see Section 5.1).

Highlights of the library are described below, with examples of use in Caubweb.

4.2.1 Asynchronous Execution

Most of the library follows an asynchronous, non-blocking model of execution. This model follows from the event-driven nature of Tk and Tcl's fileevent (see also Section 6.1). Applications such as Caubweb will have many partial operations in progress at any one time. This model means that each ongoing operation will get a chance to accomplish some work when it is able. However, each operation must gracefully relinquish control after doing some small amount of work.

Cobweb requires any method that is not short-lived and non-blocking to use either a *callback* or a *continuation* to guarantee liveliness of the application. A callback is a small script that a caller passes to a method, after which the caller is expected to relinquish control. Upon completion of the request, the callback has a return value appended and is then evaluated. This results in control being returned to the caller. The callback will occur in the context of the facility to which the caller made the request, meaning that the caller cannot assume anything about its own state. An example use of callback in Cobweb is to initiate an asynchronous request to an HTTP server and later receive a handle for the connection.

A continuation is used by a component to schedule its own, long running operation. A continuation is similar to a callback but adds some state that the caller passes in and then later receives as part of the upcall. An example use of continuations is to maintain the state of a weblet retrieval while other operations gain and relinquish control.

4.2.2 Web-centric Modules

URL: this is an obTcl class that can manipulate URL syntax. It can parse a URL string into component parts, as well as reconstruct it with a `tostring` operation that allows the use of a *base* URL object. This is used to resolve relative URLs.

HTTP Protocol: this is an obTcl class with a simple interface and several stackable implementations. Http uses several structures to track individual connections. These structures include the message (msg), connection (conn) and HTTP header (header). Built into the module is the logic for making outgoing requests directly or via a proxy.

The basic interface includes methods such as MakeReq (which uses a callback) and Close. Implementors of the interface are modules that an application will invoke, causing the modules to initialize themselves into a protocol graph. For instance, if an application just initializes the Http and HttpProxy modules along with a server loop, the result will be an HTTP tunnel (that is, a cacheless proxy). If it also initializes the HttpCache module, then it becomes a caching proxy.

HTTP Server: This includes a server loop and a simple, *mock* HTTP server. The server loop enables a socket listening on a port or ports to process incoming requests. Each request is broken apart into command and target and then dispatched to the handler for server using that socket. The HTTP server reads disk files and returns their contents to the remote side. It understands several special CGI-like file types. One is called *htcl*, which is Tcl code to be evaluated in a slave interpreter, the output of which is sent to the remote. Another is *thtml*, which is an HTML file with embedded Tcl code that is expanded via subst. *htcl* is used to create the Caubweb control panel pages, which include dynamically updated data from Caubweb's state.

HttpCache: This is an obTcl class that configures itself *above* Http in order to transparently trap requests from the higher levels and take appropriate action based on the state of the cache.

HTML Parser: This is a variant of the HTML parser from sntl [23] combined with changes from SurfIt [1], which are based upon Steve Uhler's html_library.tcl[30]. It has been heavily modified so that it is re-entrant and can parse incrementally. Additional work went into cleanly splitting the main logic apart so that it is no longer driven to render the HTML it parses. Finally, it can be told to parse against tagsets, so that a document can have particular information located in it. Caubweb uses a tagset to find rendering elements (e.g., <IMG>), hyperlinks (e.g., <A HREF>) and other assorted references.

Weblet: This module implements the "weblet walker," which is responsible for identifying and fetching into the cache all members of each weblet. It makes substantial use of work list and queue management abilities.

5. Tcl's Strengths and Weaknesses

Arguments about the suitability of Tcl for any given project usually focus on several key concepts, such as the interpreted nature, the "everything is a string" model, its use as a glue language, and the ease of constructing user interfaces with Tk. These aspects derive directly from Tcl's roots but do not necessarily reflect the use of the language to implement complete stand-alone applications such as ours. We used Tcl to construct an entire application, which allowed us to focus on some different key aspects of the language. We present some of our observations here.

5.1 Portability

Tcl, being a scripting language, affords a natural portability of code. This concept, along with the (at that time) forthcoming Windows and Mac ports of Tcl 7.5, meant that an implementation of our system in Tcl could easily allow us to "buy into" a truly portable system. This mostly came true. To date, we have run Caubweb snapshots on Windows95 and Windows NT, most flavors of UNIX, and the Macintosh (System 7.0 and higher). In addition, the InfoPad project at the University of California, Berkeley, has used Caubweb on its base station running Unix to provide disconnected access to web pages via the InfoPad. However, support on multiple platforms required more effort than we originally expected.

Creating pure Tcl is a useful approach to avoiding machine dependencies. It means that end users can utilize scripts without work such as compiling new code. Prior to Tcl 7.5, the general rule was that a new extension had to compile its modules and then statically link with the Tcl libraries itself. This process resulted in large binaries with limited mixtures of extensions. The new **load** command resolves this problem.

However, not everything can be made with pure Tcl. Caubweb depends upon two key mechanisms (*copychan* and *host*) that are implemented in C as a minor extension. This requires us to provide source code and force end-users to compile it. We take the middle approach of also providing the pre-compiled loadable modules for some number of *supported* platforms. This still requires work to compile the module on all the platforms we support, but that happens infrequently, when we make a change in the source of the loadable module.

This work is often well worth doing. We have long believed that some Tcl/Tk applications win users over better than others based upon the convenience of installing and using the application. When the system is portable with little or no effort, a user will be inclined to use it. The Tcl community is littered with systems that have not achieved broad user acceptance, such as *tkwww* [27] and *tknews* [26]. On the other hand, systems such as *exmh* [31] match this model.

The negative side effect of using a compiled extension is that it becomes an additional portability constraint. Requiring a mechanism that is only available as a compiled extension (or some other compiled C code) adds an obstacle for the casually interested party to try out the system on an unsupported platform. This may well reduce the set of people who choose to evaluate a new system.

This trade-off was constantly evaluated during the project to decide when and what components were to be used in the system. In the end, we ended up with three loadable modules, only one of which was required by our system (copychan). The other two (*host* and the obTcl accelerator) are optional; pure Tcl code exists to implement the functionality of each module if it is not available. This allows us to get the performance gain of the loadable module when we have done the compilation work, allowing us to speed up the system when we are committed to it on a particular platform.

This trade-off was also partially responsible for us not using other extensions, like MTtcl [10] or Incr Tcl [15], although other reasons played a role for these extensions (i.e., they were not initially available for Windows and the Mac).

Even with Tcl's high degree of portability, we still had significant problems. Our first concern was (at that time) the ongoing development of Tcl 7.5. The Mac and Windows ports contained many bugs that prevented the system from being directly used. For much of 1996, we had to provide our users with bug fixes for Tcl 7.5 and later Tcl 7.6. To avoid causing our end users to acquire the tools to compile under Windows and MacOS, we also provided binaries of the fixed Tcl/Tk distribution on those platforms.

5.2 Extensibility for Performance

For Tcl, the extensibility mechanism is a direct trade-off against portability. But, when performance really matters, it is time for a compiled C extension. In the end, we took a cautious approach of only utilizing loadable modules that were either absolutely required for Caubweb to work or for which we could easily provide a workaround that was written in plain Tcl (at a performance penalty).

We added *copychan* for this reason. It is a derivation of *unsupported0*, which has become *fcopy* in Tcl 8.0. However, it has two important characteristics. First, it properly returns the number of bytes copied and, thus, never loses data. This allows us to use it reliably to copy data from a web server into a cache file and know that the cache file is correct (fcopy also has this characteristic). Second, it allows us to "multicast" the data to multiple file channels. This allows us to utilize the same data stream from a web server to create the cache file and to return the data to the user's browser. The result is that we save significant compute time doing a read/write/write from a Tcl loop. In addition, prior to Tcl 8.0, it was impossible to program that loop in Tcl at all, since there was no reliable way of storing the file data in a Tcl variable (since the data could contain arbitrary binary data).†

One area in which additional performance would have been useful was in parsing and manipulating HTML. Until the ability was added to parse HTML files piecemeal (using a trick from SurfIt! [1]), this was a major cause of perceived sluggishness in Caubweb. However, when weblet retrievals are in progress, the overhead of parsing HTML consumes the vast majority of Caubweb's compute time. Caching the hyperlink references derived from the HTML parsing may alleviate this problem.

†. Several extensions would have helped in this regard, the most interesting being the memchan extension [12].

6. Revisiting the Decision to Use Tcl

Many of the basic reasons for choosing Tcl have paid off handsomely. The language has proved suitable for constructing our system. The ease of writing has proven itself over and over, as new capabilities are easily added to the system. The learning curve for producing usable, good code for project engineers has been short.

A significant advantage has been achieved in the lack of problems related to fixed size string buffers, data structures, memory allocation, garbage collection, and stray pointers. These common problems have plagued numerous systems, including other web systems (servers and browsers).

In addition, Tcl mechanisms sometimes lead to particularly elegant implementation techniques. The best example of this is the weblet template language, which is internalized into a form that can be directly evaluated by the Tcl interpreter. As a result, for very little cost we have created a powerful and expressive syntax for describing web page relationships without being burdened with building the expression evaluation engine.

Nonetheless, some aspects of Tcl were problematic; we discuss these below.

6.1 Issues in Event Handling

Tcl has an inherent event-driven nature. This is derived partially from the historic implementation of Tk and windowing systems, and due to bias on the part of the language designer [19]. This forces any complex system such as Caubweb to use an architecture of asynchronous execution with callbacks. This is a workable solution, but it does have drawbacks. Any operation that blocks the return to the event loop causes the application to appear to hang momentarily. There are some ways to avoid this, such as the work to make the HTML parser compute in smaller quanta. However, there are situations in which one cannot avoid blocking and for which Tcl provides no help. Hostname (DNS) lookups in the socket channel code is one example. This causes the application to become entirely unresponsive, and the user to complain. This is true for Caubweb and for other Tcl-based systems (this is a common user complaint about exmh, which even has a secondary helper process to avoid this problem).

An alternative to this approach might be to use a threaded operating system with a threads extension to Tcl and then partition the application so that significant work is done in separate interpreters, each in their own thread. This is the approach take with Audience1 [21]. By having operating system support for threads, a single thread can block in an operation and not prevent other threads from continuing execution. This naturally limits the use of the application to those systems that support this collection of abilities. Further, the model of separate interpreters causes other problems, as state (variables) can not be shared across interpreters and thus must be duplicated, with the cost being paid in the memory footprint of the system.

To contrast, even a language like Java designed with threads in mind only partially escapes this problem. Threads are supported even on systems where there is no underlying operating system support. Java forces all I/O to be asynchronous but does not force the event-driven model on the programmer. Java actually goes the other way; whereas I/O is asynchronous, the I/O interfaces are not. Thus, to read on a stream, a thread is forced to block.

6.2 Tcl Changes

Our work began when Tcl 7.5 was still in alpha testing. Our initial framework used TclX 7.4 as a stopgap until 7.5 was stable enough. The process of working against a moving target has occasionally been painful. Not having a Windows port of Incr Tcl meant that any useful object-oriented extension needed to be pure Tcl so that we could use it on all our platforms. Along the way we have helped to debug several serious problems in the socket code for the Windows and Mac ports. It was not until Tcl 7.6p2 that we finally were able to stop shipping our own Tcl binaries.

This is not a complaint about the work that the Sun Tcl Group produces, but rather an observation about differing priorities. Their priorities have been different than what we would have liked. Completeness and correctness in the socket code is more interesting, *to us*, than completeness in native look and feel.

6.3 The Impact of Tcl 8.0

The worst shortcoming we have come across in our Tcl work can be summed up in two quick points: the need for extensions to support basic language operations (OO and megawidget paradigms, binary I/O) and the overall performance of an interpreted system. These have been identified by many people in the past [21]; as of this writing, Tcl 8.0 has been released in beta form, and a new age is upon us.

To be sure, it does not solve all the problems. However, performance measurements show some marked improvements in some areas, new I/O mechanisms will allow arbitrary data to be manipulated without the potential loss of information, and a common core mechanism for namespaces opens the door to a unified OO mechanism.

These potential benefits come with a price. Our system will need substantial changes to work with the new incompatibilities introduced with these new mechanisms. For instance, obTcl uses the familiar "::" separator between class and instance, but this is now usurped by the namespace facility. Consequently, none of our existing obTcl-based code can work with Tcl 8.0b1. With obTcl now no longer being maintained, we must revisit the decision to use obTcl. Will Incr Tcl be adapted to the new namespace mechanism quickly?

In addition, while *fcopy* replaces some of the mechanism of *copychan*, it does not handle multiple destinations. Hence, there will still be a need for our own modified versions.

6.4 Call to Arms for the Tcl Community

When reviewing features that different languages have to offer, one notices that Tcl is missing a rich, cohesive, organized, standard library. The community, while building many fine components and libraries, does not have a mechanism in place to collect those components in a single library structure. Tcl code is too often reinvented, ignoring the advantages that comes from reusing existing software [24].

The Tcl contributed archive is a start at constructing such a library. However, much of the code in the archive is out of date and little of it can be expected to work together. The archive is exactly that, a static collection of what has been done rather than a unifying repository against which applications can be built.

In contrast, other language environments come with these components. Java has the JDK (and more). Perl has a rich set of primitives plus CPAN [3], a well organized library of additional code. CPAN is so successful that a single implementation of a module tends to be adopted by the community; further, changes by the community are often actively merged back into the library.

We believe that Tcl needs an archive like CPAN. The existing contributed Tcl archive includes reusable code, but the cost of reuse is almost always too much work. A single, cohesive library would mean there would not be five different HTML parsers or four different HTTP implementations, and that Tcl developers could create new and interesting things rather than spending time reinventing these same old pieces.

Tcl, to date, has avoided providing much in the way of auxiliary library code. Tcl 8.0 breaks from that tradition in a significant fashion by providing an new *http* package that implements some of the most useful client-side HTTP operations. This could be the start of a new, useful extended library, if only other existing useful code is incorporated.

Additionally, the core needs to embrace other sorely missing operations as Tcl language constructs. We believe there is a need for lower level bindings to a complete set of POSIX C library functions. This mechanism would support *scripts* in accomplishing work that currently requires a loadable module written in C.

When considering such a community library, one also needs to consider what might be expected from the Tcl core to empower reusable code. With the support of namespaces, an encapsulation mechanism for the benefits of data hiding can become standard. A full object-oriented environment is not necessary, but a mechanism like Incr Tcl's *ensemble* or the *major/minor* extension alone might be sufficient. Other noteworthy items include the framework changes for megawidget support and the core hooks to provide useful debugger support (such as single stepping).

6.5 Contributed Core Changes

The evolution of the Tcl core has progressed too slowly, in contrast to Java where change happens too quickly. This forces the Tcl developer community to create needed infrastructure themselves. If the core eventually incorporates this mechanism, it typically uses an incompatible implementation. By taking the slow approach to incorporate these changes, the effect of making the developer's code out of date is that the community always has new hurdles to overcome just to keep current with Tcl.

7. Conclusion

We have developed a substantial server-based application, targeted to multiple platforms, using Tcl. We selected Tcl for platform-portability, extensibility, and ease of distribution, and it satisfied those criteria relatively well. The extensibility turned out to be particularly important because of deficiencies in evolving versions of Tcl that were either platform-specific or related to core functionality. We identified strengths and weaknesses of Tcl that stood out during our work, and we challenged the Tcl community to work together to create a rich, cohesive, standard library.

The Caubweb application uses a set of components (Cobweb) which we have already re-used in an ancillary application called CaubView. It is based upon HistoryGraph [8][9], the result of another project in our organization that used Tcl. HistoryGraph was a visualizer that allowed a user to dynamically view relationships among web pages retrieved by the user's browser. A revised version of HistoryGraph serves as a prototype for viewing relationships within the Caubweb cache. Revisions included adding new relationship calculations and replacing some prototype components with more robust equivalents from Cobweb, such as the HTML parser, the URL class, and the HTTP protocol modules. Incorporating the Cobweb components took less than one week.

Availability

The latest version of Caubweb (including the Cobweb library), instructions for its use, release notes, and user mailing lists are available for our web page:

http://www.opengroup.org/RI/www/dist_client/caubweb/

Caubweb is covered by distribution terms that allow free use and redistribution of the system or Tcl source for non-commercial research and evaluation purposes.

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under the contract number F19628-95-C-0042. The views and conclusion contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Stavros Macrakis identified some of the initial important concepts in the Distributed Clients project. Charlie Brooks developed the merge-back aspects and substantially improved CaubView, the initial version of which was contributed by Scott Meeks, based on the HistoryGraph work of Meeks, Frederick Hirsch, Charlie Brooks, and Murray Mazer [8]. Charlie Brooks, Frederick Hirsch, Susan LoVerso, Mary Ellen Zurko, and the workshop reviewers provided useful feedback that improved the contents and presentation.

References

[1] S. Ball, "SurfIt! A WWW Browser," *Proc. Fourth Annual Tcl/Tk Workshop,* Monterey, CA, USA, July 1996, pp. 161-171.

[2] C. Brooks, M.S. Mazer, S. Meeks, and J. Miller, "Application-Specific Proxy Servers as HTTP Stream Transducers," *Proc. Fourth International World Wide Web Conference*, Boston, MA, USA, December 1995, pp. 539-548. http://www.w3.org/pub/Conferences/WWW4/Papers/56/

[3] Comprehensive Perl Archive Network (CPAN), http://www.perl.org/CPAN/CPAN.html

[4] R. Fielding et al., "Hypertext Transfer Protocol - HTTP/1.1," RFC 2068.

[5] P. Floding, patrik@dynas.se, obTcl 0.57b3, ftp://ftp.dynas.se/pub/tcl/

[6] Forefront Technologies. http://www.ffg.com/whacker/whacker_tech.html

[7] J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification*, Addison-Wesley, 1996.

[8] F.J. Hirsch, W.S. Meeks, and C.L. Brooks, "Creating Custom Graphical Web Views Based on User Browsing History," *Sixth International World Wide Web Conference*, Santa Clara, CA, USA, April 1997. http://www.opengroup.org/RI/www/waiba/papers/www6/hg.html

[9] F. J. Hirsch, "Building a Graphical Web History Using Tcl/Tk," Fifth Tcl/Tk Workshop, Boston, MA, USA, July 1997.

[10]S. Jankowski, booga@netcom.com MTtcl documentation, ftp://www.neosoft.com/pub/tcl/sorted/devel/MTtcl1.0.tar.gz

[11]R.H. Katz, "Adaptation and Mobility in Wireless Information Systems," *IEEE Personal Communications*, 1 (First Quarter 1994), pp 6-17.

[12]A. Kupries, a.kupries@westend.com, "Memory Channels in Tcl," ftp://ftp.westend.com/pub/aku/memchan1.2.tar.gz

[13]M.S. Mazer et al. "Issues in Mobile Computing Systems: Guest Editor's Note," *IEEE Personal Communications, Special Issue on Mobile Computing*, December 1995, pp. 12-13.

[14]M.S. Mazer and J.J. Tardo, "A Client-side-only Approach to Disconnected File Access," *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, December 1994, pp 104-110.

[15]M. J. McLeannan, "[incr Tcl]: Object-Oriented Programming in Tcl," Proc. of the Tcl/Tk Workshop, University of California at Berkeley, CA, June 10-11, 1993.

[16]J. Miller, P. Resnick, and D. Singer. "Rating Services and Rating Systems (and Their Machine-Readable Descriptions)," *World Wide Web Journal*, Vol. 1, No. 4, Fall 1996, O'Reilly and Associates, Inc.

[17]The Open Group Research Institute, Project Overview for Distributed Clients, http://www.opengroup.org/RI/PubProjPgs/dist_clients.html

[18]The Open Group Research Institute, Project Overview for Intelligent Browsing Assistance for the WWW, http://www.opengroup.org/RI/PubProjPgs/waiba.html

[19]J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[20]J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)", *Proc. 1996 USENIX Technical Conference*, January 1996.

[21]A. Sah, et al., "Programming the Internet from the Server-Side with Tcl and Audience1^{TM}," *Proc. Fourth Annual Tcl/Tk Workshop,* Monterey, CA, USA, July 1996, pp. 183-188.

[22]M. Schickler, M.S. Mazer and C. Brooks, "Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web," *Proc. Fifth International World Wide Web Conference*, Paris, France, May 1996. http://

www5conf.inria.fr/fich_html/papers/P15/Overview.html

[23]S. Shen, slshen@lbl.gov, Sam's New Tcl Library 0.4, ftp://www.neosoft.com/pub/tcl/sorted/devel/sntl-0.4.2.tar.gz

[24]H. Spencer, "How to Steal Code -or- Inventing The Wheel Only Once", *Proc. Usenix Conference Winter 1988,* Dallas, TX, USA, February 1988, pp. 335-345.

[25]T.H. Romer et al., "The Structure and Performance of Interpreters," *Proc. ASPLOS VII*, Cambridge MA USA, October 1996.

[26]tknews news reader, version 1.2b, ftp://www.neosoft.com/pub/tcl/sorted/net/tknews.1.2b/tknews.1.2b.tar.gz

[27]tkWWW World Wide Web browser, version 0.12, http://www.mit.edu:8001/afs/athena.mit.edu/course/other/cdsdev/html/welcome.html

[28]Traveling Software, http://www.gowebex.com/

[29]L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl (2nd Edition)*, O'Reilly and Associates, Inc., 1996.

[30]B. Welch, S. Uhler., "Tcl/Tk HTML Tools," *Proc. Fourth Annual Tcl/Tk Workshop,* Monterey, CA, USA, July 1996, pp. 172-182.

[31]B. Welch, exmh mail reader, http://www.smli.com/~bwelch/exmh/

Jacl: A Tcl Implementation in Java

Ioi K. Lam, Brian Smith
Department of Computer Science
4130 Upson Hall
Cornell University
Ithaca, NY 14853-7501

{ioi, bsmith}@cs.cornell.edu

Abstract

Jacl, Java Command Language, is a version of the Tcl [1] scripting language for the Java [2] environment. Jacl is designed to be a universal scripting language for Java: the Jacl interpreter is written completely in Java and can run on any Java Virtual Machine. Jacl can be used to create Web content or to control Java applications.

This paper explains the need for Jacl as a scripting language for Java and discusses the implications of Jacl for both the Java and Tcl programming communities. It then describes how to use Jacl. It also explains the implementation of the Jacl interpreter and how to write Tcl extensions in Java.

1. Motivation

One on-going question in the Tcl community is, how can Tcl exploit the popularity of Java and the World Wide Web. There are two projects that try to bring Tcl into the world of Java and WWW. The Tcl Plugin [3] allows the execution of Tcl scripts inside Web browsers. However, the Tcl Plugin runs only inside certain browsers (Navigator and Explorer), requires the user to install software on local machines and does not communicate well with Java. Tcl-Java [4] allows the evaluation of Tcl code in Java applications, but it requires native methods and thus cannot run inside most browsers.

A Tcl implementation in Java will facilitate the creation of portable Tcl extensions [4]. Tcl is a portable scripting language. However, although Tcl provides some support for writing portable extensions, maintaining Tcl extensions written in C for multiple platforms is still a difficult task, especially if network or graphics programming is involved. Currently Tcl runs on more platforms than Java. However, due to the large number of commercial Java developers, Java will probably catch up in the near future and run on more platforms. If Tcl implementations can be written in Java, the Tcl community can leave the portability issues to JavaSoft and other Java implementers and concentrate on developing the Tcl core interpreter and extensions.

On the other hand, Java needs a scripting language as powerful as Tcl. Java is a structured programming language and is not a good scripting or command language [7]. Currently, scripting languages that can be used on Java platforms, such as Javascript and VBScript, are proprietary, non-portable and restrictive. Javascript and VBScript run only on the browsers that support them. Their scripting engines are system-dependent and cannot run on arbitrary Java Virtual Machines. These languages are good for scripting HTML pages, but they lack the features that would allow their deployment at any larger scale. For example, Javascript cannot define new classes; Java applets cannot directly pass events to VBScript [5, pp. 843]. Moreover, these scripting languages are not embeddable and thus cannot be used to control Java applic ations.

Jacl is a comprehensive solution to the problem of Tcl and Java integration. Since the Jacl interpreter and extensions are written completely in Java, they can run inside any JVM, making Tcl an embeddable, universal scripting language for Java. By using the Jacl interpreter, Java programmers can use Tcl to control simple Web pages, complex networked Java applications, and anything in between.

2. Using Jacl to Script Java Applications and Applets

Java applications and applets and very similar to each other. The following section concentrates on applets only but the discussion holds true for Java applications as well.

```
button .b1
button .b2
button .b3

.b1 config -text "    ..............fastest............        "
.b2 config -text "    ..............faster.............        "
.b3 config -text "    ...........not so fast...........        "

pack .b1 .b2 .b3

proc scroll {btn time} {
    set str [$btn cget -text]
    set str [string range $str 1 end][string index $str 0]
    $btn config -text $str
    after $time scroll $btn $time
}

scroll .b1 100
scroll .b2 200
scroll .b3 500
```

Example (2.1)

2.1 Using Jacl in an Applet

The Java classes that implement Jacl are in the cornell.* hierarchy. The conell.applet.Shell class can be used to execute Jacl scripts inside applets. The following HTML code shows how to embed an Jacl-enabled applet inside an HTML page:

```
<applet width=300 height=100>
  code=cornell.applet.Shell.class>
  <param NAME="jacl.script"
         VALUE="buttons.tcl">
</applet>
```

When the cornell.applet.Shell class starts up, it will create a Jacl interpreter to execute the script file specified by the jacl.script parameter.

2.2 Using Tcl and Tk Commands

Jacl supports all the basic Tcl commands (e.g., `string` and `puts`, as well as the control constructs such as `if` and `for`.) It also supports a subset of the Tk commands for building graphical interfaces. Example 2.1 shows a script that performs a simple animation by scrolling text across three buttons at different speed. This script should look familiar to experienced Tcl/Tk programmers because its syntax is exactly the same as traditional Tcl/Tk programs. Figure 2.2 shows how the applet appears inside Netscape.

2.3 Accessing Java Classes with Raw Scripting

There are two ways for Jacl scripts to access Java classes: *Raw Scripting* and *Custom Scripting*. Raw scripting uses the Java Reflection API [8] to directly create Java classes and invoke their methods and fields. The following example shows how an applet can use raw scripting to manipulate a java.lang.Date object:

```
set date [new java.lang.Date]
button .day -text [$date getDay]
```

The `new` command is used to create an instance of a Java class with the given name (in this case, java.lang.Date.) The `new` command returns an *object command*, which can be used to invoke the methods of the object. In the above example, the `getDay` method of the object is called to query the current day of the week on the system.

The object command supports two special options, `get` and `set`, to query and modify the fields of the Java object. In the following example, we create an object of the java.util.Vector class, add several elements and the query the `elementCount` field to determine the number of elements in the vector object:

```
set vector [new java.lang.Vector]
$vector addElement "string1"
$vector addElement "string2"
set num [$vector get elementCount]
```

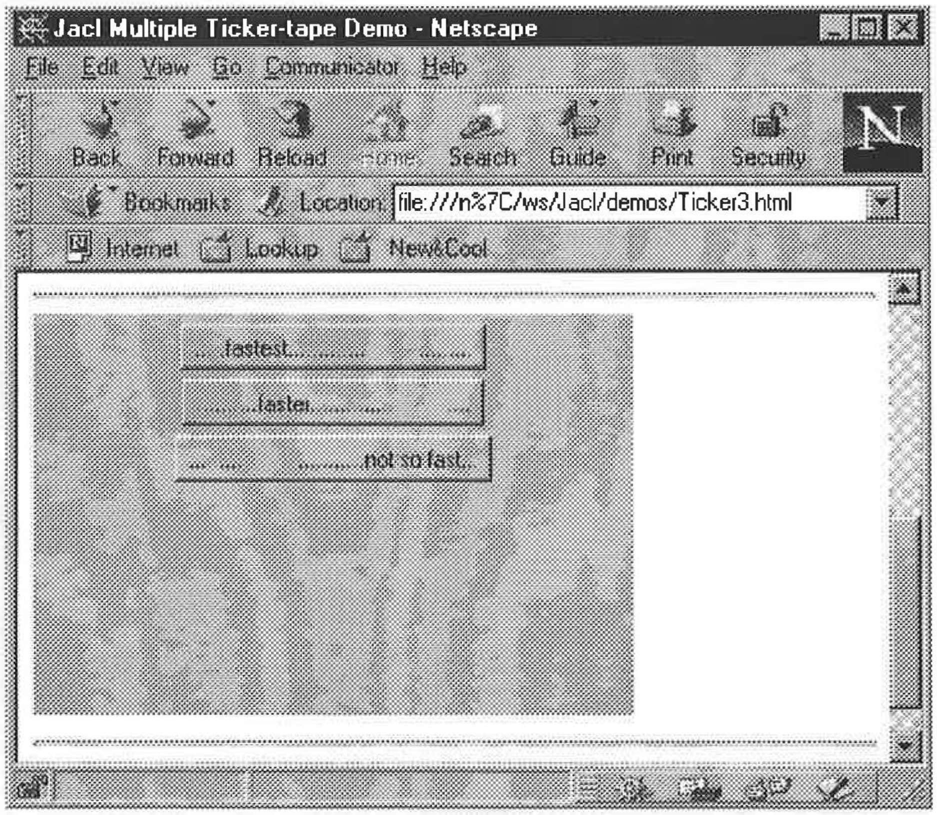

Jacl Multiple Ticker-tape Demo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security
Bookmarks Location: file:///n%7C/ws/Jacl/demos/Ticker3.html
Internet Lookup New&Cool
...fastest...
...faster...
...not so fast...

Figure (2.2)

When the methods and fields of the Java objects are invoked, Jacl will coerce the parameters when necessary. For example, in the following code segment, the parameters passed to the `setSize` method of the frame object may be represented as strings in the script. Jacl will convert them into integers before invoking the `setSize` method:

```
set frame [new java.awt.Frame]
$frame setSize 100 200
```

Jacl uses a set of heuristics to disambiguate the invocation of overloaded methods. For example, if we have a Java class with an overloaded method `foo` that can take either an integer or a string parameter:

```
class A {
    void foo(int i);
    void foo(String s);
}
```

and we manipulate this class with the following script:

```
set obj [new A]
$obj foo 1
$obj foo abcd
```

The first call to `foo` will invoke the integer version because the parameter looks like an integer. In contrast, the second call will invoke the string version because it is not possible to convert `abcd` to an integer.

In the cases where the disambiguation heuristics are insufficient, one can use *method signatures* to choose which version of an overloaded method should be called. A method signature specifies the name and argument types of a method. For example, the following code forces the string version of the `foo` method to be called even though the argument looks like an integer:

```
set obj [new A]
$obj {foo String} 1
```

2.4 Custom Scripting

In custom scripting, Jacl scripts access Java objects through a scripting API provided by a Jacl extension (see section 4 for a discussion on writing Jacl extensions.) The `button` command in section 2.2 is an example of a custom scripting API for accessing java.awt.Button objects. One can access Java objects through raw scripting or custom scripting. Figure 2.3 shows the differences between raw- and custom scripting and compares the scripting code with Java code.

As shown in figure 2.3, both raw- and custom scripting provides interactive access to Java classes. Custom scripting has the advantage of supporting a more convenient syntax but it requires the writing of Jacl extensions. Therefore, raw scripting is generally used to gain "quick and dirty" access to Java objects. When it is necessary to have better scripting support for Java objects, Jacl extensions can be written to provide a custom scripting API.

3. Implementation of the Jacl Interpreter

The Jacl interpreter is based on the Tcl 7.6 interpreter. Most of the parsing routines for Tcl scripts and expressions are translations of the Tcl 7.6 C source code into Java code. Therefore, the Jacl interpreter is compatible with the Tcl 7.6 interpreter. In fact, the Tcl 7.6 test suite is used to ensure that Jacl parses and executes scripts in exactly the same manner as Tcl 7.6.

There are two major enhancements in Jacl with respect to Tcl 7.6: object support and exception handling. These enhancements improve efficiency and simplify the implementation of the Jacl interpreter and extensions.

3.1 Object Support

In Tcl 7.6, all objects are represented by strings. In Jacl, however, an object can be represented by any Java object. For example, in the following code:

```
set a 1234
incr a
```

After the first line, the variable `a` will contain a string "1234". At the second line, the `incr` command will coerce the string into an integer and then increment its value by one. After this operation, the variable `a` will contain a integer with the value 1235.

Moreover, lists in Jacl are implemented as copy-on-write Vector objects to improve both access time and storage efficiency. In the following code

```
set list1 [list 1 2 ... n]
set c [lindex $list 3]
set list2 $list1
...
...
lappend list2 abc
```

the `lindex` operation takes constant time, compared to the O(n) time in Tcl 7.6. Also, after the `set list2 $list1` command, the two variables `list1` and `list2` will refer to the same object. The contents of the list will be copied into the `list2` variable only when a destructive operation,

Coding Method	Program Listing	Interactivity	Simple Syntax	Extension Required

Java Code	`Button b = new Button("Hello");`
	`Color c = new Color(255, 255, 0);`
	`b.setForeground(c)`
	`add(b);`
		No	No	--

Raw Scripting	`set b [new Button Hello]`
	`set c [new Color 255 255 0]`
	`$b setForeground $c`
	`$applet add $b`
		Yes	No	No

Custom Scripting	`button .b -text Hello -fg #ffff00`
	`pack .b`
		Yes	Yes	Yes

Figure (2.3)

such as `lappend`, is applied to that variable.

3.2. Exception Handling

Another difference between Tcl 7.6 and Jacl is how they handle error conditions. Tcl 7.6 uses return code such as `TCL_OK` and `TCL_ERROR` to indicate the success or failure of script execution. The Tcl 7.6 C source code spends considerable efforts in checking the return code of functions. In contrast, Jacl uses the Java exception mechanism to handle runtime errors. Thus, the Jacl source code is less cumbersome than the Tcl 7.6 C source code. For example, inside the Tcl parser, where errors can happen in many sections of the code, the Jacl implementation uses about 30% fewer lines of code than the Tcl 7.6 implementation written in C. Figure 3.1 compares the coding style between Jacl and Tcl 7.6

4. Writing Jacl Extensions

A Jacl extension is generally a collection of new Tcl commands. A Tcl command is a class that implements the `Command` interface. The command can be added to a Jacl interpreter by passing an instance of its class to the `CreateCommand` method. Example 4.1 shows how a `print` command can be defined

One interesting feature of Example 4.1 is the way arguments are passed to `CmdProc`, the command procedure. Because the arguments passed to a command may be Java objects of any type, it is no longer sufficient to pass the arguments as `(int argc, char ** argv)` in Tcl 7.6. Instead, Jacl passes the arguments in a `CmdArgs` object. The following code shows the interface of the `CmdArgs` class:

```
public class CmdArgs {
    public int argc;
    public String argv(int index);
    public int intArg(int index);
    public double doubleArg(int index);
    ....
    public Object object(int index);
}
```

A command can use the converter methods, such as `argv`, `intArg` and `doubleArg`, to convert the arguments into the required types. The command can also use the Java `instanceof` operator to directly infer type information of the arguments. In example 4.2, the `index1` command verifies that it receives an non-empty Vector object as its first argument before returning the first element of this Vector.

5. Status and Future Directions

As of this writing, the Tcl parser, expression evaluator and most basic Tcl commands have been implemented in Jacl. It also supports a subset of the Tk commands for creating graphical interfaces. Jacl is already being used to create simple applets to run inside browsers. It can also be used to control Java applications and applets with raw- and custom scripting. A beta release is expected to be available in the third or fourth quarter of this year.

Many more features have been planned for Jacl, including built-in debugging, supports for multi-threading, and a byte-code compiler. To find out more about the new developments of Jacl, please visit the Jacl home page at http://www.cs.cornell.edu/ home/ioi/Jacl.

Acknowledgment

I would like to thank Thomas Breuel and Anil Nair for providing valuable inputs during the early design stage of Jacl. Thomas sent me the basic design of the `cornell.Tcl.Command` interface, which I put into Jacl without much change. Scott Stanton and Jacob Levy were instrumental in the design of the raw scripting API.

Java code	C equivalent

```
...
int i = interp.GetInt(string);
  // exception is thrown if string
  // doesn't contain a valid integer
...
```
	```
	...
	int i;
	if (Tcl_GetInt(interp, string, &i) != TCL_OK) {
	        return TCL_ERROR;
	}
	...
	```

Figure (3.1)

```
import cornell.Tcl.*

class PrintCmd implements Command {
    Object CmdProc(Interp interp, CmdArgs ca) throws EvalException {
        if (ca.argc != 2) {
            throw new EvalException("wrong # args: should be \"" + ca.argv(0)
                   + " string\"");
        }
        System.out.println(ca.argv(1));
        return "";
    }
}

....
    // Create a new "print" command.
    interp.CreateCommand("print", new PrintCmd());
....
```

Example (4.1)

```
class Index1Cmd implements Command {
    Object CmdProc(Interp interp, CmdArgs ca) throws EvalException {
        if (ca.argc != 2) {
            throw new EvalException("wrong # args: should be \"" + ca.argv(0)
                   + " vector\"");
        }
        if (!ca.object(1) instanceof Vector) {
            throw new EvalException("expected Vector but got \"" + ca.argv(1)
                   + "\"");
        }
        Vector vector = (Vector)(ca.object(1));
        if (vector.elementCount < 1) {
            throw new EvalException("Vector must not be empty");
        }
        return vector.elementAt(0);
    }
}

....
    // Create a new "index1" command.
    interp.CreateCommand("index1", new Index1Cmd());
....
```

Example (4.2)

Bibliography

[1] John Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Massachusetts, 1994

[2] Ken Arnold, James Gosling, *The Java Programming Language*, Addison-Wesley, Massachusetts, 1996

[3] Jacob Levy , *A Tcl/Tk Netscape Plugin*, Proc. of the 1996 USENIX Tcl Workshop, Monterey, 1996.

[4] Scott Stanton and Ken Corey, TclJava: Toward Portable Extensions, Proc. of the USENIX 1996 Tcl/Tk Workshop, Monterey, 1996.

[5] Michael Morrison, et al., *Java Unleashed*, Sams.net Publishing, Indianapolis, 1997

[6] Brian Lewis, *An On-the-fly Bytecode Compiler for Tcl*. Proc. of the USENIX 1996 Tcl/Tk Workshop, Monterey, 1996.

[7] John Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, http://www.sunlabs.com/people/john.ousterhout/scripting.html, 1997.

[8] Sun Microsystems, Inc., *Java™ Core Reflection, API and Specification*, http://java.sun.com/products/jdk/1.1/docs/guide/reflection/, 1997.

A Typing System for an Optimizing Multiple-Backend Tcl Compiler

Forest Rouse

ICEM CFD Engineering,
Berkeley, CA 94705.

Wayne Christopher

ICEM CFD Engineering,
Berkeley, CA 94705.

Abstract

This paper describes the typing system used by the ICE 2.0 Tcl compiler[Rouse]. The typing system tracks the usage of variables and allows the compiler to reduce the number of instructions required to carry out Tcl instructions. It will also allow future improvements in code emission by making it possible to carry out standard compiler analyses and optimizations[Aho].

1 Introduction

Tcl has become a popular scripting language[Ousterhout]. In connection with the rise of the internet as a popular medium for information transfer, there is a high probability that it will continue to grow in popularity. One of the advantages of Tcl as a scripting language is the lack of explicit typing. Everything in Tcl is a string, therefore the complexity of both the programs written in Tcl and learning the language itself are reduced dramatically. Unfortunately, the simplicity of "everything is a string" makes optimizing compilation extremely difficult.
Additionally, the semantics of the language are not formally defined but rather given by the implentations of the various commands. Hence, different commands can and do treat language elements differently. For instance, the command

```
set a(1) 3
```

treats the variable `a(1)` differently than the command

```
foo $a(1)
```

The variable `a(1)` is parsed differently in the two cases. There is no difference between the two methods of parsing the variable name most of the time, but in cases like

```
foo $a(this is an example)
```

it does.

Finally, Tcl commands can cause non-local side-effects on variables. The act of placing a trace on a variable and the actions on upvar and uplevel variables can cause any cached knowledge about the variable's value type to become invalid. Any typing system must be able to handle all possible non-local effects.

Early tests with the ICE 1.0 Tcl to C compiler reveals some interesting statistics. We use a simple ASCII file format converter as a benchmark example. This converter has about 430 lines of commented Tcl code. The code has a high proportion of string manipulation and relatively few arithmetic manipulations.

The act of compiling control statements, recognizing Tcl words at compile time, and not copying argument strings to invoked procedures results in a factor of 3 speed-up. The final version of the 1.0 compiler has a factor of 7.5 speed-up on the same benchmark. The 8.0a1 version of the Sun byte compiler[Lewis] results in a factor of 4 speed-up. Hence, the current compilers (the Sun on-the-fly and the ICE 1.0-1.2 Tcl to C) to within a factor of two rely on parsing the statements once (recognition of Tcl words), and not copying strings when invoking procedures. We must go to other optimizing techniques to get larger speed-ups.

Placing constraints on the variable type and possible side-effects allows the compiler to avoid emitting code to carry out traces or unnecessary type conversions. It will allow the compiler to safely remove unnecessary code in the near future. This paper describes the typing system, the related system of "promises", and future optimizations based on the typing system.

1.1 Overview of Compiler

The ICE Tcl compiler is an ongoing commercial project started in 1994. It was first designed to emit only C code. The Tcl to C translation has proved useful to our customers because of the improved performance of the Tcl code along with the ability to hide their intellectual property. Obviously, shipping just the Tcl code opens the program up to both customers or anyone else with access to a computer system.
However, most Tcl users like the convience of rapid turn-around available using the Tcl interpreter. The ICE 2.0 compiler has been redesigned so that instead of emitting just C code, the compiler emits an intermediate language that can be translated to multiple backend languages. The first language that will be available will an enhanced Sun 8.0 byte code. This version is entering tests as of this writing. We expect that both Java and C backends will be available by September of this year.
Users can "mix-and-match" emitters to tailor their application. That is, users can have specific procs compiled to "C" while others compile to any other possible backend including bytecode.
The compiler builds a parse tree. The nodes of the parse tree represent Tcl language elements. During synthesis, code representing the language element is emitted. The synthesis of both typing and overall script attributes are also directed by the node. This design makes it considerably easier to maintain and extend the compiler than the 1.0 version.
The principle upgrade to the 1.0 system has been the implementation of a C++ node to represent variables. Variables are tricky because different statements parse variables differently. This design allows all language elements to treat variables in the same way.

2 The Typing System

We can describe the typing system to first order as a set of attributes whose values at any point in the program is the current Tcl type that is associated with every recognized Tcl variable. This description is incomplete only because we must also follow all allowable side-effects. Hence, the type attribute includes whether or not the variable has a known type, locality (global, local, upvar, uplevel, or argument), whether or not the variable has been unset, whether or not the variable is possibly being traced, whether or not the variable is in a loop, and whether or not the variable is an array indexed with a constant string, an array indexed with a variable string, or is a run-time computed variable name, which we call a *dynamic* variable (as in **set $x 3**).
Essentially the action of the typing system is to properly synthesize these attributes after every state change of the system. We can avoid emitting type conversion statements if the variable is known and is of the correct type. Other optimizing actions can also be taken on some known variables. This reduces the number of statements required in a compiled Tcl program thereby reducing the execution time.
The difficulty comes from the fact that every Tcl statement must be analyzed to determine the possible side effects it might have.
Consider, for example, the following sequence:

```
proc a {} {
    set c 3
    unset c
    set x [list a b c d e]
    set y [list set unset]
    uplevel 1 {
        unset a
        [lindex $y 1] [lindex $x 1]
    }
}
```

This sequence unsets variables "a", and "b" in the scope of the caller, and the variable c in the scope of proc a. Trival analysis is required for understanding the effects on the variable c. Calling procedures must analyze the effect of the proc to determine the effect of unsetting "a". Finally, to determine that the variable "b" is unset requires constant folding along with the calling procedure determining the effect of unsetting "b".
The current type analysis system in the ICE 2.0 compiler currently carries out analyses in the local scope of the procedures. Recursive typing, or the ability of the caller to analyze the effects of a procedure on variables in its own local scope and constant folding through variables are currently being tested. In lieu of recursive typing and constant folding, the current type system states whether a given procedure unsets remote variables. This invalidates the variable cache in all calling procedures. Any read or write of the variable subsequent to the invalidation of the cache will require that the variable will be read via the standard procedure "LookupVar".
All Tcl statements that have an unanalyzable effect invalidate the cache in similar way. All commands of the form "$x a b c..." (both on command lines and in control statements), eval statements, and variable traces can cause the variable cache to

be invalidated. Users can make "promises" – compiler directives that limit the effect of unanalyzable statements. These compiler directives are not suitable for every Tcl program. For example, a user cannot make a promise about statements computed at run–time in a Tcl script that calls the following procedure:

```
proc a {x} {
    uplevel {
        set c "unset $x"
        eval $c
    }
}
```

The reason is that the "eval" statement has the side effect of invalidating the variables in the calling scope.

2.1 Promises

The typing system is conservative. The worst case scenario is used to synthesize attributes for unanalyzable statements. Users can constrain the effect of unanalyzable statements thorough the use of "promises". Promises can be made about the possible side effects of

- catch statements with variable bodies,
- dynamic statements (`$x a b c ..`),
- eval statements and control statements with variable bodies,
- trace procedures (effect of variable traces), and
- uplevel statements with variable bodies.

Statements can be deemed "safe", "nounset", and/or "notrace". Users invoke compiler directives on the command line as in

```
Tcl_compiler -filescript -safe trace foo.Tcl
```

If the compiler directive is invoked in this manner, the directive applies to the entire Tcl script in the file foo.Tcl. Additionally users can also direct that a specific procedure scope be compiled with a specific promise as in

```
proc -safe trace a {a b c ...} {
  <proc body>
}
```

In this case, the "safe trace" directive applies to procedure a and all procedures defined within its scope.

The "safe" promise directs the compiler to assume that no statement in that category unsets a variable, changes the type of the variable, or establishes an "unsafe" trace on a variable. The "nounset" promise just directs the compiler to assume that no statement in that category unsets a variable. The "notrace" promise directs the compiler to assume that no statement in that category establishes a trace on the variable.
The promise that has the largest effect on any program is the promise that variable traces are "safe". Since every variable (including variables local to a proc), the compiler must assume the worst about every variable reference. The compiler analyzes if a variable local to a proc has a trace on it. But all non-local variables potentially have a trace placed on the variable.
Consider the following code:

```
proc a {} {
    trace variable a r trace_a
    proc trace_a {name element op} {
        set $name "foo"
    }
    b
}
proc b {} {
    uplevel 1 {
        set a 3
        incr a
    }
}
```

When compiling procedure b, seemingly the variable "a" in procedure b could be typed as an integer and a string after the **set a 3** statement. Unfortunately, the following incr statement will invoke a read trace that will cause the variable "a" to be changed to the string "foo". The type of "a" as an integer must instead be changed to unknown before we compile **incr a**. The following is the code emitted for the incr statement in procedure b:

```
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "read");
if (__tempReturn_i != TCL_OK)
                   goto __tempLabel_1;
__tempReturn_i = Tclc_CheckVar(
    interp, a, "a", NULL);
if (__tempReturn_i != TCL_OK)
                   goto __tempLabel_1;
__tempDummy_i =
    Tcl_ConvertUnknownToNumeric(
        interp, &a);
if (!(a->varAttr.typeAttr&TCL_TYPE_INTEGER)) {
    __tempReturn_i =
        Tcl_ConvertFromUnknownToDString(
            interp, &a);
    if (__tempReturn_i != TCL_OK)
                    goto __tempLabel_1;
    Tclc_ErrIncrVar(interp,
        a->value.allValue.dString.string);
    __tempReturn_i = TCL_ERROR;
    goto __tempLabel_1;
}
TCL_ONLYVALIDVAR(a, 2);
a->value.allValue.integer += 1;
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "set");
```

All of the code prior to `a->value.allValue.integer += 1;` is to assure the validity of "a" and that the variable can be promoted to an integer.
However, if the read or write trace does not modify the type of the variable as in

```
proc a {} {
    trace variable a r trace_a
    proc trace_a {name element op} {
        global c
        lappend c "$name $element $op"
    }
    b
}
proc b {} {
    uplevel 1 {
        set a 3
        incr a
    }
}
```

the script can be compiled with the promise of "safe trace". The compiler will instead know the type of "a" as an integer is still valid. The following code is emitted for the incr statement in proc b:

```
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "read");
if (__tempReturn_i != TCL_OK)
    goto __tempLabel_1;
TCL_ONLYVALIDVAR(a, 2);
a->value.allValue.integer += 1;
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "set");
```

A factor of three fewer lines of C code are emitted in this case. We eliminate the check on the validity of the variable "a", and the promotion of "a" to an integer and the checks that both operations succeed. That is, since we know any possible trace on the variable "a" cannot have a side effect that modifies the type or validity of "a", we can avoid the checks that assure the state of variable "a".

Test	ICE 1.0	ICE 2.0	Speedup(%)
Loop	102	68	50
10! (using loop)	98	76	28
Sum	1958	648	200
List	17651	10669	53
Reverse list	28443	21663	31

Table 1: Comparison of the execution speeds of C code produced by the Tcl 1.0 compiler and the Tcl 2.0 compiler on a variety of benchmarks. Times are in μsec.

2.2 Tcl Lint

The in depth analysis of Tcl programs has led us to release "Tcl Lint". This program is the ICE 2.0 compiler with no code generator. The compiler with its ability to determine the possible side-effects of statements allows us to create error or warning messages when a variable is used before it is set. When there is no possibility for a variable to be set, an error is reported. If an unanalyzable statement is executed prior to variable usage, a warning message is generated if the proper user modifiable warning level is set.
Additionally, the compiler checks that usage of core functions and user defined procedures matches what is expected. Currently, we check that the number of arguments matches the number of arguments given, and for core functions, the usage of the function matches what is expected. Extensive analyses of catch, for, foreach, if, regexp, regsub, switch, and uplevel are carried out.
Tcl Lint can warn of unanalyzable statements. Statements like `uplevel $a` and `switch $a $b $c $d ...` can be pointed out to the user. Tcl Lint can also warn of procedures that use the implicit return mechanism that are expected to actually return a value. The compiler can slightly improve the code emitted if it knows that a procedure does not actually return a value. Finally, control commands with empty bodies are pointed out.

3 Current Status

Table 1 shows the speedups of the 2.0 compiler over the 1.0 compiler. The benchmarks were carried out on a Solaris 5.0 machine. We see that over a number of different types of benchmarks, the effect of

having a typing system is relatively modest. The speedups range from 25% to 50%. The benchmark of summing numbers between 1 to 1000 achieved factor of 3 speedup over the ICE 1.0 compiler is the exception to this rule.
There is also a decrease in the amount of emitted C code. We were able to emit a factor of 3.5 less C code, compile to a factor of 2.5 less object code and get a factor of 1.5 reduction in the executable size with our standard benchmark. This reduction occured in part to the fact we could use the promise of "safe trace" in the benchmark.

4 Future Plans

The typing system is a prerequisite to all forms of standard compiler optimizations. We must be able to analyze the side effects of moving or eliminating code before we can carry out the optimizations. Hence, we are now in the position to implement the following standard compiler optimizations:

- recursive typing,
- constant folding,
- common subexpression elimination,
- loop strength reduction, and
- code motion.

The effect of these optimizations will be script dependent.
We expect to have an early version of the 2.0 compiler with a bytecode emitter ready by the time of the conference. It will emit C code and an extended set of Sun bytecodes. The extended bytecode set will take advantage of our typing information. We hope to be able to present preliminary timing studies of emitted bytecode at the conference.
Finally, we are seriously thinking of using the compiler to emit Java bytecodes. One can regard the "C" code emitter as a model for all other static language emitters. The only real technical challange to emit Java, therefore, is a reasonable user usage model. We most certainly can emit bytecodes for the joint Java–Tcl interpreter and we believe that a suitably constrained pure Java port of many Tcl core functions could be written in a reasonable length of time.

5 Conclusion

We have described a multiple backend compiler for use in compiling Tcl. Additionally, it has a typing system that allows the compiler to infer all possible side effects of Tcl statements. This improves the code emission, speeds up the target code, and makes it possible to include standard compiler optimization techniques in near future to automatically improve current Tcl scripts.

6 Acknowledgments

Our thanks to OpenMarket Corporation who partially funded this work.

7 Availability

Both the ICE compiler and Tcl Lint are available for electronic downloading at

`ftp.dnai.com/users/i/icemcfd/tcl`

or visit our web site at

`http://www.icemcfd.com/tcl/ice.html`

Users can request a demo license by downloading, uncompressing, and untarring the software. Then just execute the command

```
tcl_compiler -hostid
```

and send the resulting information to

`tcl@icemcfd.com`

References

[Rouse] Forest Rouse and Wayne Christopher, "A Tcl To C Compiler", Proceedings of the 1995 Tcl/Tk Workshop (1995), Toronto, Ontario, Canada., July 1995

[Aho] Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman, *Compiler Principles, Techniques and Tools*, Addison-Wesley, 1988

[Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).

[Lewis] Brian T. Lewis, "An On-the-fly Bytecode Compiler for Tcl", Proceedings of the 1996 Tcl/Tk Workshop, Monterey, CA., July 1996

TclOSAScript - Exec for MacTcl

Jim Ingham
Lucent Technologies
(now at Sun Microsystems)
jim.ingham@eng.sun.com

Raymond Johnson
Sun Microsystems
rjohnson@eng.sun.com

Abstract:

We describe the TclOSAScript extension to MacTcl. TclOSAScript provides the ability for MacTcl scripts to run scripts in any other OSA compatible language on the Macintosh. Since the OSA is the standard mechanism for interapplication communication on the Mac, this allows MacTcl to run other applications, and provides an exec like facility (though arguably using a much richer communication model.)

I) An Introduction to the Open Scripting Architecture

The usual interapplication communication mechanism for Unix based tools relies on the simple expedient of connecting the standard in and standard out channels of the spawned process to channels of the parent process. The communication between the parent and child processes then mimics the user interaction with the process, namely typing commands on the command line. The "exec" command in Tcl, as well as the open-with-pipe command, use this mechanism to give Tcl access to processes in the surrounding system.

This method, unfortunately, is inapplicable on Macintosh systems. Macintosh applications are only implemented with a Graphical User Interface (GUI). There is no command line, and thus no concept of standard in and standard out. However, this does not mean there is no interapplication communication mechanism. Instead, a much richer method of scripting external tasks is provided by the Open Scripting Architecture (OSA)[1,2].

This problem, and the OSA solution presented here, were first mentioned in papers in the Usenix Tcl/Tk 95 and 96 conferences [3,4]. There are two implementations of this solution that have been presented. Ted Beldung wrote a extension called ASTcl[5] which only worked with AppleScript, and did not use many of the advanced features the OSA offers. TclOSAScript, which was developed concurrently, is a more complete implementation. This is the one we will detail in this paper.

There is a real problem that the OSA aims to solve as well – beyond the fact that there may well not be a text based command interface to tasks in a modern GUI operating system. For while the command-based form of communication is easy to implement, to actually drive an application you need to know the particular language that the application uses, be it simple command line switches, or a real language such as Tcl. There is no generic way to expose the functional elements of an application without tying them to a particular scripting language.

The solution is to define the basic objects and verbs that the application supports in a language neutral way. Then the scripting language must provide a mechanism for querying out these elements. Finally, the operating system will provide a messaging system that will allow any scripting language to drive the objects of the target application.

a) Specification of Verbs & Objects – the aete

The specification of objects is achieved with the *Apple Event Terminology Extension*. This is a resource in the resource fork of the application which lists the 'events' to which the application responds, and the classes of objects in the application.

Examples of events that an application might recognize are the '`get`', '`set`' and '`create`' events, which just provide access to the objects in the application. These form part of the "Standard Suite" of commands that all well designed OSA applications should support. There are also more specialized verbs, such as '`select`' or '`revert`' in a Word Processor app or '`download`' and '`view file list`' in the FTP client application Fetch. Events also take parameters, such as which word to select in a word processor, or which file to download in Fetch.

Examples of objects are the windows of an application, or the words and paragraphs in a word processor window. There is also a way of specifying a containment hierarchy, so each object can be contained in other objects, and can contain other objects, . Finally, the objects may take qualifiers. An example of a moderately complicated object specifier, in AppleScript dialect, is: "`the file "mactcltk-full-8.0a2" in the transfer window "ftp.sunlabs.com" of the application "Fetch 3.0.2"`".

The aete makes the connection between the text phrase which specifies an object or verb, and some packed 4 character code which will occur in the compiled form of the interapplication messages. It also specifies the parameters for the verbs (with the notion of required and optional parameters), and the containment hierarchy of the objects. Once you know the aete of an application, you know the form any message to that application must take.

b) The Messaging system – Apple Events

The messaging system in the OSA is governed by the Apple Event Manager. An Apple Event is a data structure that describes an event in terms of the aforementioned packed 4 character codes. It starts with the main verb, and then usually has a direct object with its qualifiers, and perhaps other parameters. Apple Events support nesting, so that a single event can support recursive evaluation (much like the [] syntax in Tcl).

One application under the MacOS sends an Apple Event to another application by building up the Apple Event data structure, and then passing it to the Apple Event Manager, which places the event in the event queue of the target application. The target application then handles this event by parsing up its contents and performing the required task.

c) OSA compliant languages

All interapplication communication is carried out, under the hood, by the exchange of Apple Events. However, this is too low level an exchange mechanism to be useful to the average user.

The job of an OSA compliant language, then, is to wrap a more human syntax around the construction of the Apple Event messages. The approved method for doing this is to read the aete of a target application, and import the verbs and nouns found therein into its own syntax in a natural way. Done this way, the aete can serve as documentation for how to script each language, and the language can dynamically appropriate new applications as they are encountered.

Of the two major OSA compliant languages, only AppleScript[6] dynamically reads the aete of the target application. UserLand Frontier[7] has a static glue table for each application that relates the event and object codes with commands in the Frontier language. This limits the number of scriptable applications that Frontier can actually drive.

d) OSA Components

The final part of the OSA architecture is the OSA component. This is an implementation of the notion that a scripting language is not so much a part of the application, as a service that is provided to applications, to aid them in exposing their operative parts.

Tcl has been very successful precisely because it makes it easy to bind together the functional elements of the application without having to recompile the application. But to make use of Tcl, the application must embed a Tcl interpreter within the object code of the application. This ties the use of the application to Tcl; you cannot dynamically choose which scripting language you want to employ.

The OSA goes further to make the operating system the manager of scripting solutions. The provider of a scripting language writes some glue code that registers the scripting language with the operating system. Applications request a connection to the scripting component, and then send script data to that component to be evaluated. The script data can even contain tags identifying its language, so that the application can receive scripting data in any of the available languages, pass it to the OSA component manager, which will in turn route it to the appropriate component.

This connection can be used in two ways. One is to allow an application to drive other applications, so that it does not have to duplicate their functionality. A good example of this is the MicroSoft Internet Explorer, which hands off e-mail, FTP and NNTP requests to the user's favorite e-mail, FTP and Usenet clients using the OSA.

A deeper use of the OSA is to "factor" the user interface elements of the application from its core functionality. The user interface elements do not directly call the subroutines that do the work of the application. Rather they send AppleEvents back to the application, which receives these events, and then dispatches them to the appropriate internal routines.

The advantage of this is that you can then "attach" scripts to the UI elements, which scripts do the work of dispatching the AppleEvents. This immediately provides an architecture much like HyperCard, or the Tcl/Tk callbacks, but with the added benefit that the callback language is left up to the user's discretion.

II) Tcl and the OSA

There are three steps to fit Tcl into the OSA architecture. The first is to allow Tcl to use the services offered by the OSA components installed on the system. This is the function of the TclOSAScript extension which is described in the rest of this paper. The next two steps have not yet been completed. The first of these is to provide a Tcl command to build and dispatch Apple Events. The final step is to install Tcl/Tk as an OSA component in its own right, so that Tcl would be an option in all the major Scripting development environments.

TclOSAScript

The first stage of this incorporation is completed. The TclOSAScript extension allows Tcl scripts to connect to any available OSA component, and send scripts off to be evaluated by that component. Currently connections to both of the popular OSA languages, AppleScript and UserLand Frontier have been tested. TclOSAScript will be included as a shareable library in the 8.0 release of MacTcl. What follows is a general description of the TclOSAScript extension.

1) The OSA command

The first step in using any OSA component is to get a connection to an instance of the component. At startup, TclOSAScript scans the list of OSA components, opens a connection to each one that is found, and creates a Tcl command to access that component. So if the user has AppleScript installed on their machine, an `AppleScript` command will be created. If UserLand Frontier is running at startup, a `UserLand` command will be created, and so on.

There is also a generic command, `OSA`, that will allow other connections to be opened, or closed, and will allow the user to query the list of components. This is useful, for instance, to open connections with other languages.

An example of this use is communication with the UserLand language. The UserLand OSA component is installed only while the Frontier application is running. So if you wanted to use Frontier, you could open it (by using the `AppleScript` command), and then use the `OSA` command to open a connection to the UserLand component...

It is currently not useful to open multiple connections to the same component. Since the calls to the OSA components are synchronous, having several connections to a single OSA component does not gain anything.

2) Compiling and Executing Scripts:

Once you have an open component, you can send scripts to it, to be executed. The simplest way to do this is with the `execute` subcommand (see fragment 1):

```
AppleScript execute {
    tell application "Fetch 3.0.2"
        download (url "ftp://" & ¬
              "ftp.sunlabs.com" & ¬
              "/pub/tcl/mac/"   & ¬
              "mactcltk-full-8.0a2.sea.hqx")
    end tell
}
```

Fragment 1

```
set getNewFiles [AppleScript compile {
    tell application "Fetch 3.0.2"
        open (url "ftp://ftp.sunlabs.com/pub/tcl/mac")
        set thisWin to (transfer window "ftp.sunlabs.com")
        set nfiles to (count remote item in thisWin)
        set retVal to {}
        repeat with i from 1 to nfiles
            if the modification date of (remote item i of thisWin) > refDate then
                download (remote item i of thisWin)
            end if
        end repeat
    end tell
    get retVal
}]
AppleScript execute {set refDate to (date "Saturday, February 1, 1997 3:19:02 PM")}
button .b -text "Check for files" -command "AppleScript run $getNewFiles"
pack .b -padx 6 -pady 6
```

Fragment 2

This will execute a script that uses the Mac application "Fetch" to get the 8.0a2 version of mactcltk from the Sun ftp site...

However, complicated scripts can take some time to compile, so if you want to run the same script over and over, you might want to compile the script once, and then run it many times. For this purpose, there is a `compile`/`run` pair of subcommands.

In the example in Fragment 2, we make a button whose action is to download all the files in the /pub/tcl/mac directory whose date is later than the some reference date (chosen pretty much at random here...):

The "`AppleScript compile`" command passes the script to AppleScript to compile, and returns a script handle for the compiled script. This handle can be passed to the "`AppleScript run`" command, which will execute the script, and return whatever value the script returns.

3) Other Tricks

We have also used another trick of TclOSAScript, to define the variable refDate. AppleScript executes its scripts in *"Script contexts"*, which are just namespaces that retain all the variables and procedures that are defined in them. The `AppleScript` command opens a default context which it uses for all script execution. So the "`AppleScript execute`" line in Fragment 2 will set the `refDate` variable which the `getNewFiles` script uses in its execution.

Note that although the notion of a script context is a part of the OSA specification, it is not one of the required parts. So, for instance, UserLand Frontier does not use them. It has its own mechanism for persistence (the object database).

Because Tcl and the code within the AppleScript commands are just strings, we can combine the two languages for even more power. As an example let's extend the above example to always use the current time instead of a set date. The following code uses double quotes to allow Tcl based substitution to occur before the `AppleScript` command compiles the string into AppleScript byte codes.

The example in Fragment 3 will get the current seconds and format the time into a string that is acceptable for the AppleScript `date` command. You must be careful, however, when doing such combinations to make sure you create a valid AppleScript command. A good understanding of Tcl's quoting conventions is required.

```
AppleScript execute "set refDate to (date \"[clock format \
[clock seconds] -format \"%A, %B %d, %Y %X %p\"]\")"
```

Fragment 3

4) Return Values

One other issue is the return values that come back from AppleScript. Suppose that we want to ex-

```
...
set scriptRsrc [AppleScript load $scriptFile]
AppleScript run $scriptRsrc
...
button .f2.view -text "Display Files" -command loadList

proc loadList {} {
       global hostName filePath refDate scriptRsrc
       .f1.lf.lb delete 0 end
       AppleScript execute -variable retList -context $scriptRsrc \
           "listMoreRecent(\"$hostName\",\"$filePath\",date \"$refDate\")"
       eval .f1.lf.lb insert 0 $retList
}

button .f2.load -text "Download" -command {getFiles}

proc getFiles {} {
       global hostName filePath scriptRsrc
       foreach index [.f1.lf.lb curselection] {
             set name [.f1.lf.lb get $index]
           AppleScript execute -context $scriptRsrc \
               "getFiles(\"$hostName\",\"$filePath\",\{\"$name\"\})"
       }
}
```

Fragment 4

pand the following example to query the /pub/tcl/mac directory, and present a list of new files to the user, so that she can choose which ones to download. Then we could write a subroutine in AppleScript that returned a list of new files.

However, this will be returned as an AppleScript formatted list, not a Tcl Formatted list. Now sometimes you may need the AppleScript format (e.g. to pass back to AppleScript). At other times the Tcl format is more appropriate.

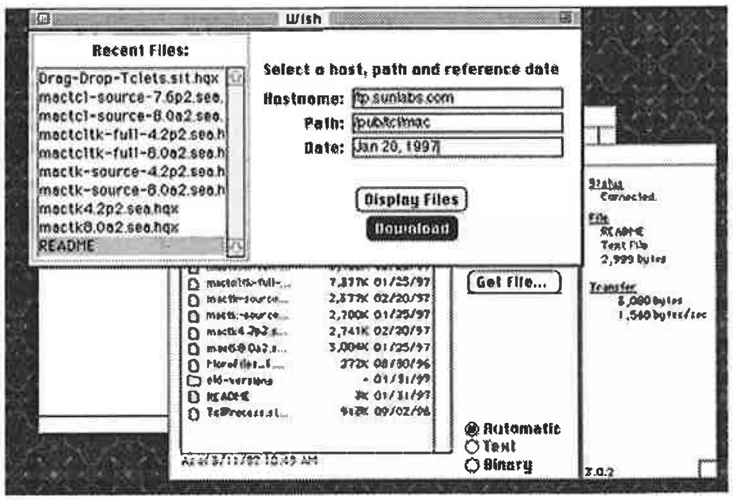
In TclOSAScript, the return value of the run command is the AppleScript form of the result. Then we have added a "`-value`" flag to the run and execute commands. You use it to pass the name of a variable to the command, and TclOSAScript will parse AppleScript lists up into Tcl lists, and AppleScript Records into Tcl arrays, and put the result in that variable.

5) Loading Scripts

Finally, Tcl is not the most convenient site for developing AppleScript code. Rather than have to cut and paste from the Script Editor into Tcl, TclOSAScript has a "`load`" command that will load script data from a script resource (which is the format all the AppleScript development environments write out), either in the application itself, or an auxiliary file.

This facility allows for useful methods of user customization for Tcl applications. For instance, a MacTcl application could read the script resources out of all the files in a "Scripts" folder, and populate the "Scripts" menu in the application with them.

Using a stored script, containing two AppleScript subroutines *listMoreRecent* and *getFiles*, we can create the application shown in Figure 1 and Fragment 4, which will display the new files in the given directory, and download the ones chosen in the list. In typical Tcl fashion, the whole application is 60 lines of Tcl, and 24 lines of AppleScript code...



Drag-Drop-Tclets.sit.hqx
mactcl-source-7.6p2.sea.
mactcl-source-8.0a2.sea.
mactcltk-full-4.2p2.sea.h
mactcltk-full-8.0a2.sea.h
mactk-source-4.2p2.sea.h
mactk-source-8.0a2.sea.h
mactk4.2p2.sea.hqx
mactk8.0a2.sea.hqx
README

Hostname: ftp.sunlabs.com
Path: /pub/tcl/mac
Date: Jan 20, 1997

Figure 1

III) Future work

The TclOSAScript extension is almost complete. It would perhaps be useful to provide an asynchronous mode of execution (analogous to `exec` with an `&`) Another major enhancement would be to make each component command run in a separate thread. Then it would be useful to open a connection to, say, the AppleScript component, send it a long-running script. Then open another connection if you have some work to be done before the first task is completed.

Aside from these details, there remain two parts of the complete incorporation of Tcl into the OSA scheme. The first remaining step is to provide a facility to parse the aete of target applications, and a Tcl based mechanism for dispatching Apple Events based on the verbs and objects found therein.

Parsing the aete is relatively straightforward. However, defining a Tcl representation for AppleEvents will require more work. The nesting of objects that is required to specify elements in a target application can require the construction of Apple Events of considerable complexity. For instance, we might want to query out "Every paragraph of the first window of the application "WordPerfect" whose font is "Times Roman" and whose first word is "Foo"". The equivalent Tcl syntax will have to be rich enough to mirror that.

The last step is to provide Tcl as an OSA component. A proof of concept implementation was done by Vince Darley[8], but this was without Tk, and did not address many issues that will be faced by a full implementation.

IV) Summary

The work that we have done so far, the TclOSAScript extension, has given Tcl the ability to take advantage of the other applications in the Macintosh environment. Our example just used the FTP client program Fetch, but there are many other powerful scriptable applications available, including FileMaker Pro, Quark Express, and both Netscape and MSIE, to mention just a few. This richness is now available to the writers of MacTcl applications.

Conversely, the OSA scriptor now has available the Tk GUI toolkit which has proved a great boon in the UNIX world. Many of the work flow applications that have traditionally been written in AppleScript and Frontier can now be augmented with the sort of sophisticated front-end that can so easily be created in Tcl/Tk. With the native look-and-feel of version 8.0, the power of Tcl/Tk will prove as valuable on the Macintosh as it has in UNIX.

References

[1] Apple Computer
Inside Macintosh - Interapplication Communication
Addison Wesley Publishing Corporation, 1993

[2] Dave Mark
Ultimate Mac Programming
IDG Books WorldWide Inc, 1994

[3] Johnson, R. and Stanton S.
"Cross Platform Support in Tk"
In Proc: Usenix Tcl/Tk Workshop, Toronto, Ontario, Canada, 1995.

[4] Ingham, J.
"Tcl/Tk as an OpenDoc Scripting Part"
In Proc: Usenix Tcl/Tk Workshop, Montery, California, 1996.

[5] Ted Belding
<Ted.Belding@umich.edu>
ASTcl.
http://www-personal.engin.umich.edu/~streak/ASTcl-1.0.sea.hqx

[6] Danny Goodman
Danny Goodman's AppleScript Handbook
Second Edition
Random House, 1994

[7] Dave Winer
UserLand Frontier
http://www.scripting.com/frontier/

[8] Vince Darley
OSATcl
http://www.fas.harvard.edu/~darley/Vince-Downloads.html

Redesigning Tcl-DP

Mike Perham, Brian C. Smith, Tibor Jánosi
Cornell University
Ioi Lam
Sun Microsystems

Abstract

Tcl-DP is a loadable module for Tcl that adds advanced communication features to Tcl/Tk. Tcl-DP supports communication by serial links, IP-multicast, TCP, UDP, and email, contains a remote procedure call (RPC) mechanism, and supports the design of new protocols using modules called filters. Tcl-DP 1.0 [Smi93] was released four years ago and has since been used for numerous commercial and academic projects. With age, however, the code became so brittle that adding new features and porting to new versions of Tcl was nearly impossible. Furthermore, many of Tcl-DP's original features were incorporated in the Tcl core, making them redundant in Tcl-DP. Hence, we decided to write the latest version of Tcl-DP (version 4.0) from scratch. In this paper, we describe the new features of Tcl-DP 4.0, its architecture and implementation, and problems we encountered with Tcl's new I/O system.

1. The Tcl-DP API

The Tcl-DP API contains two main parts: commands for general communication and command for remote procedure call (RPC). Figure 1 demonstrates Tcl-DP's RPC mechanism by showing how to implement a network-wide server for generating unique identifiers. The top part of the figure shows the code that is executed on the server, the bottom part shows the code that is executed on the client.

On the server machine:

```
% package require dp
4.0
% dp_MakeRPCServer 1944
tcp0
% set i 4
% proc getID {} {
        global i
        incr i
}
%7
```

On the client machine:

```
% package require dp
4.0
% dp_MakeRPCClient host.foo.com 1944
tcp0
% set id [dp_RPC tcp0 getID]
5
```

Figure 1: A Simple ID server

The first command executed on both client and server is the Tcl `package` command, which makes Tcl-DP library functions and commands available in the current Tcl interpreter. The server executes the `dp_MakeRPCServer`[1] command, which creates a socket that is waiting for a client to connect. Finally, the server defines the `getID` command, which generates and returns a unique identifier when called.

The client connects to the server using the `dp_MakeRPCClient` command, which returns a handle that can be used to communicate to the server. Finally, the client invokes the `getID` command on the server using the `dp_RPC` command. This causes a message containing the command to be evaluated to be sent to the server, where it is evaluated and the results returned[2].

Tcl-DP also contains commands to support basic network communication. Figure 2 shows how to use Tcl-DP's communication facilities to transfer the file `/etc/motd` from a client machine to the server (alvin.cs.cornell.edu). Realize that everything in this example can be done with native Tcl. `dp_connect tcp` is semantically equivalent to Tcl's `socket` command.

After loading Tcl-DP using the `package` command, the server uses Tcl-DP's `dp_connect` command to create a TCP socket that is waiting for a connection on

[1] All Tcl-DP commands begin with "`dp_`"
[2] Tcl-DP also allows commands to be invoked on the server using `dp_RDO`, which sends the message to the server but does not wait for a return value. `dp_RDO` is useful for invoking commands solely for their side effects.

On the server (alvin.cs.cornell.edu):

```
% package require dp
4.0
% set s1 [dp_connect tcp
         -server true
         -myport 1944]
tcp0
% set s [dp_accept $s1]
tcp1
% set f [open /tmp/motd w]
file0
% while {![eof $s]} {
       puts $f [gets $s]
}
% close $f
% close $s
```

On the client:

```
% package require dp
4.0
% set s [dp_connect tcp
         -host alvin.cs.cornell.edu
         -port 1944]
tcp0
% set f [open /etc/motd r]
file0
% while {![eof $f]} {
       puts $s [gets $f]
}
% close $s
% close $f
```

Figure 2: Simple network communication

port 1944. The server then issues the dp_accept command, which blocks until a client connects, and then returns a handle for the newly connected client. Once a client is connected, the server opens /tmp/motd, copies the data sent over the channel into it, and closes the file and socket.

Note the use of the dp_accept command. We felt that it was more intuitive to connect each client explicitly rather than hardwire every server socket to simply listen. This also provides a type of serialization so that the server is not flooded with connect requests.

The client connects to the server using dp_connect, giving the hostname and port number of the waiting server. After connecting, the client opens /etc/motd, copies it to the channel, and closes the file and socket.

Note that once a channel is opened, it is treated as any other file: it can be read from, written to, or checked for the EOF condition (which, in the case of TCP, means the connection has been closed).

On the server:

```
% package require dp
4.0
% set s [dp_connect serial
         -device serial1]
serial0
% set f [open /tmp/motd w]
file0
% while {![eof $s]} {
       puts $f [gets $s]
}
% close file0
```

On the client:

```
% package require dp
4.0
% set s [dp_connect serial
         -device serial1]
serial0
% set f [open /etc/motd r]
file0
% while {![eof $f]} {
       puts $s [gets $f]
}
% close serial0
```

Figure 3: Communication with serial connection

2. New Features

2.1 Channels

The features described above were available in earlier versions of Tcl-DP. In version 4.0, we added several new features.

In addition to TCP, UDP and IP multicast, DP 4.0 supports communication via serial ports and email. No matter what its type, a channel is opened using the dp_connect command and supports the same input/output operations. The mechanism allows one channel to be substituted for another in most programs – only the channel creation and configuration options (issued through the Tcl fconfigure command) differ. For example, by changing the parameters to dp_connect and deleting the call to dp_accept in Figure 2, we can transfer the file over a serial line instead of a TCP connection, as shown in Figure 3. In the case of serial connections, the -device flag takes an OS-independent name and maps it onto the OS name. The name in the example, serial1, would map to /dev/ttya in Solaris or COM1 in Windows.

```
package require dp
set server [dp_connect tcp -server true -myport 1944]
set toclient [dp_accept $server]
set crypt [dp_connect plugfilter -channel $toclient -infilter xor
                                 -outfilter xor
puts $crypt "Boris to Natasha, we get moose and squirrel soon!"
```

Figure 4: Attaching a plugin filter

2.2 RPC

The Tcl-DP RPC mechanism has also been changed and enhanced. Recursive and out-of-order RPCs are now supported. This point will be covered in detail below. Perhaps the most important new ability is that RPCs can now be performed over any Tcl channel as long as it has been registered with the `dp_admin` command. More precisely, once a channel is created, the program can call `dp_admin register chanID`, which creates a file handler so that an inbound RPC request is automatically processed. Calling `dp_admin delete chanID` disables this behavior. The significance of this feature is that `dp_RPC` and `dp_RDO` can be performed over any channel, although `dp_RPC` should only be performed over reliable channels since packet loss can cause a program to hang while waiting for a response from the server.

2.3 Filters

Another new feature in DP 4.0 is *filters*. Filters sit between the program and the channel, transparently modifying data (e.g., encrypting/decrypting it). DP 4.0 has two types of filters: *plugin filters* and *filter channels*.

A plugin filter is designed for the common case where the data is modified using a functional interface. Encryption and decryption are examples of plugin filters. To connect a plugin filter, the programmer calls `dp_connect plugfilter`, passing in three parameters: the channel to use for input/output, and two filter functions, one for input and one for output.

Figure 4 shows how to use a plugin filter. In this example, a server accepts a connection from a client (the first three lines of code), and then attaches an exclusive-or (xor) filter to that channel to weakly encrypt the data. Any data written to the channel is transparently filtered by exclusive-or'ing it with a key (the key can be changed using the `fconfigure` command on the plugin channel). To decrypt the data, the user needs another xor plugin filter on the receiving side.

DP 4.0 comes with a uuencode/decode filter, a xor excryption filter, a hex to binary conversion filter, and a `tclfilter` plugin which allows any Tcl procedure to be used as a filter. New filters are easy to write (they require two procedures), and are registered using a C API.

The filter mechanism is general and powerful. For example, by using uuencode filters, binary data can be read from, or written to, channels. We have used filters to provide a simple command line interface to a video camera whose pan/zoom/tilt is controlled by a binary protocol over a serial interface. We believe it is possible to use plugin filters to provide APIs for secure communication, authentication, and gateways to binary RPC mechanisms.

On the other hand, certain tasks are difficult or impossible to perform using plugin filters. For these cases, filter channels are appropriate. Filter channels are complete channels with configurable options and I/O routines. A filter channel is created using the `dp_connect` command, passing in the channel type as the first argument and a flag indicating the channel to use for input/output.

To understand why filter channels are needed, consider the following scenario. A user needs to transmit datagrams over a stream channel (e.g., TCP). Since the channel has stream semantics, the datagram boundaries will be lost in transmission. In other words, if the user sends two datagrams "message1" and "message2" over the channel, the receiver might read "mess" on the first read, and "age1message2" on a second read.

A simple way to keep the datagram boundaries is to build a *packetizer*, which attaches a length field to each message. At the receiver, a *depacketizer* reads the length field and returns either a whole packet or nothing (in the case of a partial read). It is difficult to build a packetizer using plugin because they only provide functional filtering. In this case, a filter channel is used.

With filter channels, it is possible to build protocol stacks that provide, for example, flow control and guaranteed transmission over unreliable channels (any takers?).

3. Implementation

3.1 Channels

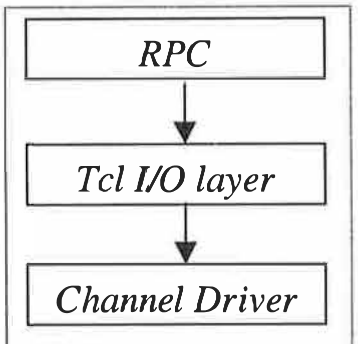


Figure 5: Tcl-DP's I/O System

Since version 7.5, Tcl has performed all I/O via *Tcl_Channels*. Figure 5 shows a block diagram of Tcl-DP's I/O system. The Tcl_Channel abstraction pushes all I/O operations into a *channel driver*, so that a generic set of operations can be performed on any communications method. For instance, to read data from a channel, one calls the `Tcl_Read()` function, which will invoke the channel driver's input method to retrieve data available on that channel. DP 4.0 provides five new channel drivers: TCP, UDP, IP multicast, serial ports and email. The `dp_RPC` and `dp_RDO` commands sit above the Tcl I/O layer, which allows us to use any channel for RPC communication.

3.2 RPC

The implementation of RPCs changed significantly in DP 4.0. The new mechanism allows RPCs to be issued over any Tcl channel, and handles duplicate or out of order return of RPCs. In addition, the protocol contains no binary data (unlike previous versions), which allows a Tcl-only implementation of RPC handlers using Tcl's socket command. These changes make the new protocol incompatible with previous versions, but we felt the new features and bug fixes justified the incompatibility.

Figure 6 shows the layout of an RPC/RDO packet. *Length* is a six digit decimal string that gives the size of the entire packet, including the length field itself. *Space* is a delimiter between fields and is simply the space character. *Type* is a one byte token character that denotes the type of this packet. For types of packets are defined: RPC ('e'), RDO ('d'), error ('x'), or return value ('r'). *ID* is a six digit decimal string that specifies an identifier for the RPC message (and its return value). It is used in processing out-of-order RPCs, as explained below. Finally, *msg* is the Tcl command to be evaluated.

For example, the command `dp_RPC tcp0 puts hi` generates in the packet

```
"     23 e      1 puts hi"
```

When a message arrives on an RPC channel, Tcl executes a file callback to process the message. This callback reads the message, buffering partial reads if necessary. It then processes the packet as follows:

- If the packet is an RPC (as determined by the *type* field), the command is evaluated. If there is an error, *interp->errorInfo* is returned to the sender in *msg*. Otherwise, *interp->result* is returned to the sender in *msg*.

- If the packet is an RDO, the command is evaluated. Nothing is returned to the sender.

- If the packet is an error code or a return value, the *id* field is used as an index into the *activation record* for the RPC, which is described below. The RPC is marked as received, the error condition set (if appropriate), and *msg* copied into the activation record.

Whenever an RPC is sent, an identifier (*id*) is generated and an activation record is created. This activation record tracks all currently executing RPCs, and allows for out-of-order and recursive RPCs. It is stored in a table, using *id* as an index.

To understand the function of the activation record, consider the following example. A Tcl-DP process sends an RPC (id=1) to host A. DP creates an activation record for the RPC and waits in an event loop for the return message. While waiting, another event (e.g., a timer event) triggers the issuance of a second RPC (id=2) to host B. Again, DP creates an activation record for the RPC and waits in a new event loop for the return message. If RPC #1 now returns, it can not be processed immediately, since the current event loop, and associated call stack, is for RPC #2. Thus, DP simply

Length [6]	Space [1]	Type [1]	Space [1]	ID [6]	Space [1]	Msg [len-16]

Figure 6: RPC packet format

records the RPC return value in the activation record associated with RPC #1. When RPC #2 returns, it is marked as received, breaking the event loop and return to the original loop. This loop exits immediately, since the activation record indicates the RPC is received.

The surprising thing about the mechanism is that its expression in C is extremely elegant. The callback for processing a return value or error message just marks the appropriate activation record. The event loop simply processes one event until the associated activation record indicates the return value has been received.

4. Open Issues

While designing DP 4.0, we ran into several problems, which we discuss in this section.

4.1 Peeking and Tcl I/O

In previous versions of DP, peeking was allowed on socket reads. Peeking allows a `read` function call to return a portion of the input available on a channel without consuming the data. It is useful when processing input messages, since a message header can be read to decide which routine should process the message.

The Tcl I/O subsystem buffers all channel input[3]. Unfortunately, this buffering also makes peeking on a channel impossible. Consider the following example: a user wishes to peek at the first 2 bytes of a 12-byte message on a UDP channel. Suppose the message is "123456789abc". Assume we `fconfigure` the socket for peeking, so that the `read()` system call does not consume the available data on the socket. When Tcl calls the channel's input procedure to read the data on the socket; it reads 10 bytes, returning 2 bytes ("12") to the caller and buffering the rest ("3456789a"). When the program attempts to read the message out of the socket, Tcl will concatenate the 8 bytes of buffered data with data available on the socket, resulting in a 20 byte message ("3456789a123456789abc").

We provide a work-around for this problem using the `dp_recv` command, which allows direct, unbuffered access to a channel's input procedure.

[3] Tcl imposes a minimum buffer size of 10 bytes on all channels. If you try to use
`fconfigure -buffersize`
to set the buffer size to less than 10 bytes, Tcl will use a 10-byte buffer.

4.2 Peeking and Tcl I/O

The `dp_RPC` command blocks until a return value is received. In DP, the RPC procedure waits in an event loop, repeatedly calling `Tcl_DoOneEvent()` until an answer for the RPC is received. Unfortunately, this also means that when an RPC message (or its return value) is lost, the dp_RPC call will never return. Since DP 4.0 allows RPC over unreliable (e.g., UDP) channels, such an event is possible. We see three work-arounds for this case. First, one can always use `dp_RDO` to remotely invoke a command. Since `dp_RDO` does not block, the program will not lock up when a message is lost. A second option is to use the `-timeout` flag of `dp_RPC`. This flag creates a timer event that marks the RPC's activation record as complete after a specified time, causing the event loop to terminate and the `dp_RPC` call to return an error. The third option is to provide a filter channel that implements retransmission for the unreliable channel, and then issue RPCs over that channel. We have not yet written such a filter channel.

5. Status and Conclusion

Because we have just finished rewriting Tcl-DP, several of the higher-level details are conspicuously missing. Foremost would be a security scheme. Tcl-DP 3.5 provided RPC checking on a line-by-line basis while, currently, 4.0 provides only a one-time RPC check. We plan to add 3.x's security scheme into 4.0 before the final 4.0 release.

Also, with the release of 4.0, we have dropped support for the nameserver in 3.3. Like DP itself, the nameserver aged poorly and we did not have the manpower to rewrite it.

In this paper, we described the new features and implementation of Tcl-DP 4.0. Tcl-DP 4.0 is a dynamically loadable version of Tcl-DP which adds several new features including new channel types, channel filters, and better RPC support. Tcl-DP is freely available from http://www.cs.cornell.edu/Info/Projects/zeno/Projects/Tcl-DP.html.

References

[Smi93] Brian C. Smith, Lawrence Rowe, Stephen Yen. Tcl Distributed Programming. In *Proceedings of the 1st Tcl/Tk Workshop*. June 1993.

Writing a Tcl Extension in Only ~~Three~~ Years
~~Four~~
~~Five~~
~~6~~ 7

Don Libes
National Institute of Standards and Technology
Gaithersburg, MD 20899
libes@nist.gov

Abstract

Expect is a tool for automating interactive applications. Expect was constructed using Tcl, a language library designed to be embedded into applications. This paper describes experiences with Expect and Tcl over a seven year period. These experiences may help other extension designers as well as the Tcl developers or developers of any other extension language see some of the challenges that a single extension had to deal with while evolving at the same time as Tcl. Tcl and Expect users may also use these 'war stories' to gain insight into why Expect works and looks the way it does today.

Keywords: Expect; Lessons learned; Software archaeology; Tcl

Expect – What is it?

Expect is a tool for automating interactive applications such as telnet, ftp, passwd, fsck, rlogin, tip, etc. [Libes95] Expect is also capable of testing these same applications. Expect is implemented as an extension to Tcl [Ousterhout]. Using Expect with other extensions is straightforward. For example, with Tk, interactive applications can be wrapped in modern graphic user interfaces allowing a program that was originally command driven to be controlled with buttons, scrollbars, etc.

Expect was first released in 1990 and rapidly became and remains today the standard tool for automating character-oriented applications. It is now used by hundreds of thousands of companies and institutions around the world. It is sold and supported by several vendors and is distributed for free from NIST[1] and with many vendor software distributions. Expect has become the de facto recommended solution for many problems faced by programmers, system administrators, and users. Expect is described in many FAQs, there are chapters on Expect in several books, and Expect is recommended by dozens of authoritative papers and articles.

1. http://expect.nist.gov

Experience Papers – Why?

It is useful to look back over lengthy periods of time. This allows observations that are uncommon. Most 'experience' papers have only a year or two to draw upon. Expect is in the unusual position that it predates almost all other extant Tcl extensions. This makes it an excellent candidate for study in the hopes of finding lessons that we can apply to more recent or even future extensions.

A second reason is that the Tcl developers are not required to consider the ramifications of their own decisions on extensions. The Tcl developers are in the enviable position of being able to change Tcl itself as needed. Needless to say, building an extension can be an entirely different experience depending upon whether you have the freedom to modify Tcl or not.

In 1995, Phelps described many difficult problems faced and solved in the TkMan and Rosetta projects during a two-year period [Phelps]. Interestingly, that paper is entirely different than this one. For instance, one of the crucial difficulties with TkMan was how to achieve high performance, something that is not an issue for Expect. ("Realistically, blazing speed is hardly needed in a program that simulates users" [Libes91].) In that and other respects, this paper is complementary to Phelps' and similar papers that offer more traditional lessons learned.

Tcl – Swamp or Savior?

It is worth pondering how Tcl affected the development of Expect. How did Tcl help Expect? Would a different approach have been better? Can these questions be answered in retrospect? After seven years of Tcl, is it possible to be objective about it? These questions are difficult to answer and for the most part, I do not have cogent answers. Although I will address these points indirectly throughout the paper, I encourage readers to try and make up their own minds as I present some of the many war stories in Expect's history.

Much of Expect's beauty is due to Tcl. Tcl really does well what it was intended to do – be a glue language. Tcl makes it easy to incorporate new functionality. Expect takes a general approach towards its problem domain. It is usually possible to solve a specific case with a specially-written program (e.g., ncftp) [Dalrymple]. The advantage of Expect is that it applies to any interactive program. So once you've learned how to apply Expect to one program (e.g., ftp), it's trivial to apply it to another (i.e., telnet, passwd, crypt).

Much of the flexibility is due to Tcl itself. Tcl provides Expect with the basic infrastructure for variables, procedures, expressions, etc. Expect need only focus on the parts of the task specific to interactions with processes. At least, that's the theory.

Tcl tries very hard not to force a particular view of the world. For example, Tcl allows OO (object-oriented) or non-OO programming – both simultaneously. And Tcl permits the addition of new control structures. In fact, I didn't even realize that's what I was doing until years later when I tried to implement Expect in other languages.[1] These are examples of approaches that Tcl encourages. And Expect follows Tcl's suggestions – except in the following cases:

- Expect got there first (i.e., there was no suggestion until later), and
- Tcl's approach was confusing, painful, or I just didn't understand it.

I will give examples of both of these as well as talk about problems caused by myself and other outside influences. Having introduced key concepts (and sagas), I will then return to cover more examples of difficult interactions with Tcl as well as other insights.

Tcl's Approach – Unpleasant or Inadequate

Some of Tcl's ideas which at first seem to fit well with Expect, did not actually fit well. This section presents several examples.

Null Strings

For many years, Tcl strings did not support nulls. This was an understandable philosophical decision. C strings were naturally null terminated, so using them simplified the C API. And Tcl as a glue language had little need for nulls. Fortunately, Expect didn't have much need either. Users don't see them – they have no printed appearance – so why would a program send nulls? In fact, there are programs with interactive interfaces that do send and receive nulls. For example, curses-based programs send nulls for screen formatting. In 1990, Expect began providing null support. Initially, it was very basic – nulls were stripped out of strings before being matched. Eventually, the support grew more sophisticated and capable, but the user interface was always unnatural due to the lack of Tcl's support.

1. I was familiar with this practice having used Lisp for many years. That's probably why I didn't think too much of it at the time I began using Tcl.

After years of begging for null support in the core, the Tcl developers have scheduled it for Tcl 8.0, expected to appear sometime in 1997 – seven years after Expect's first workaround.

Scoping & OO

Originally, Tcl's scoping rules were very limited. I found them unpleasant enough that I made Expect's commands internally look in both the local and global scopes for special variables such as spawn_id and timeout. This simplified many scripts but it was not the default Tcl behavior so it had to be explained as a special Expect feature. Had Expect been designed after many other extensions existed, I might have felt less inclined to do this, but there was no tradition being violated at the time, so I didn't think very hard about it.

A related issue is that Expect defaults to using global variables (such as spawn_id) rather than insisting that they be passed as parameters. Again, this simplifies most scripts. The vast majority of users never manipulate more than one interactive process at a time. The result is faintly reminiscent of an OO implementation. At one point, I considered implementing an object-based (OB) version of Expect in the style of Tk. But for most Expect scripts, an OB version would be less practical. Contrast a traditional Expect sequence with the OB style approach:

```
# traditional
expect "Login:"
send "$username\r"

# object-based
$telnet expect "Login:"
$telnet send "$username\r"
```

I'm not aware of any Tcl extensions that actually started with an OB orientation and later dropped it. However, CGI.pm, a Perl module for CGI scripting provides such an example [Stein]. Early releases had an OB view. While it was clean as far as scoping and namespace control, users rarely used multiple CGI objects and the OB interface was cumbersome. Thus, a non-OB/OO interface was created. In practice, CGI.pm scripts are generally a mixture of OB and non-OB/OO programming.

Pattern Strings

When I began using Tcl, I found the rules for pattern creation to be quite confusing – partly because patterns are not built into the grammar but are simply represented as strings which the pattern matcher would reinterpret. I exacerbated this problem further by using a list to provide alternation over glob patterns. This caused yet another round of interpretation, so that matching trivial strings such as "No match" had to be written: {No\ match}. Tcl 6 incorporated a regular expression engine, thereby providing alternation for free. This required the rewriting of many patterns but it was worth it, drastically simplifying things.

Tcl's general quoting conventions are still a source of misunderstanding for many. This phenomenon is common enough that people have humorously given it a name: Quoting Hell. Part of the problem is that the conventions are different than anything people are used to – such as those of the shell. Tcl and shell quoting bear a resemblance to each other but it is superficial. Tcl's quoting rules are simpler and more straightforward. For this reason, it is easy to master them – as long as a good explanation is at hand. Unfortunately, because the quoting appears so similar at first, many people prefer not to learn the rules and begin to program using another language's conventions, thereby getting themselves confused and leaving the impression that Tcl's conventions are even more illogical than the shell.

In short, Tcl's quoting hell is actually bliss. The conventions are so simple and regular that they are trivial to learn. By comparison with most other languages, Tcl syntax is a pleasure to use.

Other languages, such as Perl, take the approach that patterns are a special type. This has advantages and disadvantages. One advantage is that you can have rules specifically for pattern formation, making patterns easier to write and more readable once written. The corresponding disadvantage is that additional rules themselves can be a burden on the bulk of users who cannot afford the time or brainpower to master the large number of them [Friedl].

It is worth considering the analogy to expressions. In early releases, Tcl's expr command did not support the traditional functional notation that it does today. For example, sin($x) had to be written [sin $x]. The current expr command is much more like a traditional language, but at the same time unlike anything else in Tcl. I, personally, was not thrilled at the change. Indeed, it is widely agreed that some of the shortcuts in the expr command should be avoided. It is interesting to contrast this with Perl where the number of shortcuts is one of the attractions of the language.

Expect Got There First

In some cases, Expect's design choices were made before Tcl's. When Tcl later changed, this occasionally resulted in outright conflicts that required substantial code rewrites. In other cases, Expect's choices were incorporated by Tcl or accepted by the Tcl community as a standard solution. But in either case, the cost of leading the way was expensive.

Command Name Collisions

The first release of Tcl had no file I/O and therefore no open, close, etc. Although Expect did not do generalized I/O, "close" was a natural name for one of its functions. So when Tcl finally incorporated file I/O with a command by the same name, there was a collision. Fortunately, it was easy to tell which close was intended by a trivial examination of the arguments. In fact, Expect's close command calls Tcl's close command if that's what the arguments indicate. Similar fixes were not possible in other cases. For example, Expect's send command collided with Tk's send command and there was no way to distinguish which was intended from the arguments alone. A general solution was adopted of making all Expect commands that did not already begin with 'exp' available with an 'exp_' prefix.

After running into enough of these collisions, I eventually adopted an extremely conservative policy: Before declaring each Expect command, Expect checks whether a command already exists by that name. If so, the command is only declared with the exp_ extension. Thus, it doesn't bother Expect if other App_Inits redefine Expect's commands either before or after Exp_Init. (It is permissible to redefine commands, although for obvious reasons, this is not common practice among other extensions.) It is interesting to compare Expect to BLT [Howlett]. BLT started using the blt_ prefix on all commands in 1993 but dropped it in 1996. Besides being notationally distasteful, it was difficult to support both the blt:: and blt_ flavor for each command. BLT's author also wanted to demonstrate the need for namespace support in the Tcl core.

A related issue arose with Expect's exp_continue command. exp_continue causes the currently executing expect to restart, much like continue restarts a while loop. Both continue and exp_continue make sense (and do different things) from within an expect action. Their implementation depends upon a shared namespace which include values such as TCL_ERROR and TCL_CONTINUE. Unfortunately, Tcl provides no support for allocating these values uniquely.

Other examples of unmanaged collisions persist as well, such as zombie process identifiers. To be fair, Tcl does manage a large number of resources. Yet it is surprising that a language intended specifically for extension fails to address many areas of obvious conflict [Libes97].

Substitutions

Tk's bind command demonstrates how to obtain information using a substitution mechanism (i.e., %). This idea didn't become known to me until it was too late. By the time I became aware of Tk, Expect already used reserved variables names (e.g., expect_out, spawn_out). The % mechanism is unquestionably easier to read simply because it dramatically shortens commands.

On the other hand, substitution has its drawbacks. For instance, it is confusing when used with other extensions or commands that perform similar substitutions (e.g., format inside of a bind). One of the more flexible extensions in this regard, Oratcl, performs substitutions using @ but lets users override that character dynamically, perhaps with the expectation that yet another extension might come along and choose it too [Poindexter]. In fact, there really aren't that many 'good' characters to introduce substitutions and Tcl provides no mechanism for handling the problem in a formal way.

In any case, the 8.0 compiler may put a damper on further enthusiasm over substitutions since continually regenerated commands make it very difficult to take advantage of code previously generated by the compiler.

Event Management

In version 7.5, Tcl began providing event management, in large part taken from Tk. Before this unification, Expect had to provide its own event management for the case when Tk was not present. For example, the interact command necessarily waits for input from two sources at the same time – the user and the process. The obvious solution uses select but this doesn't work on some systems. So Expect knew how to use poll too. And on systems where neither select nor poll worked on ptys, Expect achieved the same result by using multiple processes, one to wait for each input stream. This was an ugly but standard maneuver in UNIX V7 days for management of multiple streams by any process such as cu and telnet [Nowitz].

Tk's style of event management was much more flexible than that of Expect and required significant changes. For example, consider the following code:

```
expect "password:" {
     send "$password\r"
     exp_continue
}
```

This tells expect to wait for "password:" and when found, send back a password, and to do this repeatedly. This is a typical sequence for handling a passwd-like program where it is not known in advance how many times the password will be prompted for. With Tk-style events, just about anything can happen while expect is waiting. This includes spawned processes shutting down or even being replaced with different ones. In such situations, not only could the action be inappropriate but so could the expect itself. I still see people be surprised by this flexibility.

Possibly one of the reasons this is surprising is that not all of Tcl's file operations were similarly modified to support events. For example, consider these two operations which both read a line from the standard input:

```
gets stdin
expect_user \n
```

Only the expect command allows events to be processed. In contrast, Tcl's gets command blocks all events until the gets command returns.

Needless to say, a lot of work went into Expect in order to support four styles of event management (select, poll, Tk-style, V7). And this had to be rewritten when Tcl introduced the concept of the notifier.

Solving Problems that had Nothing to do with Expect

Ferreting out bugs in Tcl itself is an example of a task that had nothing to do with Expect itself. Less obvious problems grew out of a need for functionality that really had nothing to do with Expect per se. For example, a debugger was de rigeur; however, a suitable one did not exist so I stopped Expect development to write one [Libes93]. A number of pieces of Expect fall into this same category, such as signal handling, time formatting, and others. Some of these are now available elsewhere. For example, Tcl now directly supports the clock command and TclX provides signal handling [Diekhans]. Unfortunately, at the time, I couldn't afford to wait a year or two.

Designing Software Myself (or: What I Can't Blame on Tcl)

Some of the design choices I made were poor. Most have been corrected. In a few cases, I decided it was preferable to entertain the few complaints about Expect's design over the thousands of complaints had I broken everyone's code. One example is the default timeout of expect. While the concept of an implicit timeout was innovative, it is now clear that the timeout should be off by default. Instead, expect times out after 10 seconds. Although this is changed trivially in a script, it nonetheless is something that has always bothered me. Of course, I'm equally bothered by the many other programs that also choose arbitrary time outs such as ping (20 seconds) and rsh (75 seconds). It is ironic that Expect is useful in dealing with so many other programs that have capricious timeouts. Who knows where all these magic numbers come from?

In some instances, I intentionally introduced pitfalls although I tried to do it as gracefully as possible. An example is the interpretation of expect when given a single argument. Consider:

```
expect {foo bar}
```

Expect interprets this as a request to wait for the string "foo bar". But a naive user might have intended this to mean: wait for "foo" and then execute "bar" after writing similar statements with only different formatting:

```
expect {
     foo bar
}
```

Clearly Expect has to make a guess. The heuristic Expect uses is fairly sophisticated although it is hampered by the impossibility of seeing the original quoting. In particular, the following two statements invoke Expect commands with the exact same arguments:

```
expect "foo bar"
expect {foo bar}
```

It is impossible for Expect to know how the statement originally appeared, and thus the user's implicit hint over whether the argument was a simple string or a control structure is lost. A similar problem exists with the interpretation of newline (or for that matter, any formatting character) which is represented the same way whether it was originally used for formatting or specified as an explicit string. Consider:

```
expect "\n"
expect "
"
```

When allowing this double interpretation, I knew that people would occasionally step into it, but I held my breath in hopes that the number of people would be few. And the heuristic is good enough that people have to express things really unnaturally in order to be tripped up. For example, intending "foo bar" as a control structure is extraordinarily unlikely because control structures are only useful when you have multiple patterns. With one pattern, the user isn't even going to bother embedding the action in the expect command. And if they perhaps got to one pattern-action by starting with several and deleting the others, they'll still end up with the traditional formatting (with embedded newlines). Fortunately, history has born out my optimism. In total, only three people have ever reported being tripped up by this. As a final note, the heuristic can always be avoided entirely by calling expect with the -brace or -nobrace flags. In fact, Expect does this internally to avoid recursion.

It is interesting to compare this problem with Tcl's switch command. In fact, Expect was patterned after that (nee case). One particularly unfortunate drawback of bracing an argument list was the loss of evaluation. This makes the behavior of the two different forms quite different. While the braced behavior is a natural outcome of Tcl's normal command evaluation, the lack of substitution and other features renders this form of expect near useless – a high percentage of expect patterns incorporate variable substitutions and special characters – such as "$prompt" and "\n". Thus, Expect does another round of evaluations in the style of the expr command. A modern approach to this would use the subst command but Expect still takes advantage of private Tcl interfaces simply because this work was done well before the introduction of subst (and the interfaces have continued to work).

Portability – Not!

One pleasant aspect of Tcl is the portability that it provides, both to the end-user and the extension writer. Of course, the extension writer must strive for portability as well. And that is not always an easy task. Although the primary aim of this paper is to present experiences directly relevant to the Tcl community, I feel obliged to give just one example of some of the effort that was invested in Expect to address portability in its own problematic areas. Hopefully, this material can be of use to vendors and the standards community – and amusement to others.

Many people think the existence of POSIX has solved all of our portability problems [POSIX]. Alas, it has not. There are two reasons why:

- no POSIX support on pre-POSIX systems
- POSIX doesn't standardize everything

These simplistic observations are more complex than they might at first appear. Part of the problem is that POSIX is not a simple standard. POSIX is really a family of standards – each part of which appeared at a different time. And a particular operating system can pick and choose much of which the vendor wants while still using the term POSIX to describe it. Of course, many of these 'features' can be detected using simple tests during the installation process. Alas, some vendors seem determined to make that as difficult as possible. For instance, one vendor helpfully provided all of the possible POSIX include files but for libraries that didn't exist. In fact, much of what Expect does requires that features work in a certain way interactively, so Expect must successfully compile and run many test programs in order to figure out how a system behaves. And workarounds must be provided for each missing piece or divergent behavior.

Another problem is that many vendors disable POSIX features unless specifically requested, but if POSIX features are requested, then non-POSIX extensions are disabled. And because POSIX does not cover many areas, Expect necessarily must use non-POSIX extensions on all systems. Thus, Expect never requests POSIX support in the official way.

There are a surprisingly large number of things that POSIX does not define. For Expect, one of the biggest problem areas concerns pseudoterminals, or *ptys* for short. Ptys are the operating system abstraction that allow Expect to make a process believe it is interacting with a real user at a real terminal. Unfortunately, there is no standard pty interface.

Expect supports seven major variations of pty. Some systems attempt to address portability by supporting two or even three of the variations. Of course, they are never the same two or three, and systems rarely document which are the preferred interfaces. Most of these variations have subvariations, making the number of pty interfaces over two dozen. Here are some of the pty behaviors with which Expect has to deal:

- How is a pty allocated? On some systems, the file system must be searched for them. On others, a function (e.g., getpty) is provided. There is no standard name or behavior for such functions.
- What accessibility/usability tests have to be applied to the pty? Some systems return a master but you have to successfully open it and the slave before you can really be sure it is valid. Some systems require example I/O be executed to test it as well – just opening isn't enough.
- How is the pty initialized? Are the terminal modes of the pty pre-initialized (and what are they? Can they be changed and on what basis? I.e., system-wide, session-wide? Or must we assume the pty is in some unknown state? (Some systems pre-initialize the pty, but since this isn't documented we can't rely upon it, thus we spend time re-initializing anyway.)
- Is the pty drained on slave-side close after some specified time period. Can the time be changed/disabled? Stream-implementations do this, but with varying time periods.
- Do slave side operations have to be acknowledged? Some implementations require that the master 'approve' of certain slave system calls that affect the pty.
- How does the master-side detect a slave-side close? Some systems have read() return -1 with errno = EIO. Some force the use of select. Some have select return a readable fd while some have select return an exception fd. (None do what I would consider the right (and obvious) thing with read which is return 0 like any other file descriptor.)
- What ioctl's initially have to be applied to the slave side? Some systems want I_PUSH ldterm; others also want ptem and ttcompat. Some don't want anything.
- Is setuid required? Must an entry be made in wtmp/utmp? What is the interface?
- What #includes are necessary for all of this? Each system seems to have its own unique collection of include files to describe this mess.

Obviously ptys are a quagmire. But the complexity doesn't end there. Many of the operations performed on ptys are also nonportable. This has nothing to do with ptys – these same operations are nonportable on ttys, too. For instance, there are a variety of ways to gain a controlling terminal. The usual answer seen on comp.unix.programmer is *setsid() followed by open("/dev/tty",...)*. However, this doesn't actually work on a lot of systems and there's no right or wrong because, again, POSIX leaves this undefined.

Getting the answers to questions like these is often a painful ordeal. While man pages exist for pty(4) and setsid(1), there are no man pages for interactions between them. The man pages that do exist, never provide complete explanations, leaving trial-and-error as the mechanism to finding how things really work. Of course, this is really bad. Undocumented behavior often changes from one release of the same OS to another. (Actually, even *documented* behavior often changes.) So contributed fixes that are ifdef'd for a particular vendor are often inappropriate, always suspect, and there is no trivial way to address the problem. It is depressing to receive patches from people that cannot be used because there is no assurance that it won't break support for other versions of the identical OS or even identical versions but on different platforms.

With ptys, I was forced to address each different pty interface. There was no way of avoiding it. A different solution was taken with regard to setting terminal modes. Terminal modes fall into the category of 'semi-standard'. There are well-known interfaces (two, of course) for setting terminal modes: ioctl and tcget/setattr. The latter is partially specified by POSIX. So Expect makes use of this specification when possible.

Unfortunately, POSIX only defines a subset of terminal modes. Vendors always extend the modes. In order to allow users to make use of native extensions in their familiar tongue, Expect merely calls stty after encountering anything it doesn't recognize. Calling stty is very much in the spirit of Expect which encourages reuse of existing programs whenever possible.

Unfortunately, stty cannot portably be called directly by the user for several reasons:

- Some stty implementations require redirection differently than others. BSD stty traditionally applies to the device on the standard output while SV stty applies to the device on the standard input. The obvious solution is to redirect both. Alas, a feature of modern BSD is to complain if the 'wrong' descriptor is redirected. So if the user invokes Expect's stty command with unrecognized arguments, Expect internally calls stty with the correct redirection.
- Common stty modes are interpreted differently. For instance, even though most users intuitively want to specify 'raw' mode, there is no 'standard' definition of such a mode. Indeed, some sttys don't even have such a mode.
- Window size information is nonstandard. One would think that the concept of 'rows' and 'columns' is not particularly challenging. Yet there is no standard output syntax for window size information. Not surprisingly, internally each system uses different ioctls to support it. Some systems don't support window size at all. Since window size information is so frequently used, Expect handles this itself bypassing stty. This enables scripts to be portable.

Configuration

Tcl was originally envisioned as a small language library for embedding in applications. At the first presentation on Tcl that I attended, I immediately saw the merit in this. And ignored it. Expect reversed the idea, promoting Tcl to be the important application with just a small augment of process-control commands. Viewed this way, it made sense to ignore some of Ousterhout's early recommendations, such as:

- *Include the Tcl distribution with every Tcl-application.* Expect never did this – to its benefit. As Tcl became enormously popular, it would have been silly to reship the bulky Tcl distribution over for each Tcl application. And for the most part, Expect didn't care which version of Tcl it had available. Early on, however, I did receive many complaints that Expect was too hard to use because it required people to install Tcl first.
- *Embed Tcl's version number in the application version number.* This was a reaction to the incompatibilities of different Tcl versions. Since Expect was always able to support the last several versions of Tcl, embedded Tcl's version number seemed pointless. (This may not remain true in the future, though.)

As Expect was ported from one system to another, the configuration difficulties became enormous. There were more and more configuration choices in the Makefile and users were having a hard time figuring out how to configure things. In early 1993, Rob Savoye at Cygnus Support stepped in and automated Expect's configure process using GNU Autoconf. Autoconf had problems of its own, of course, but they were minor by comparison with manual configuration. Although the effort to maintain Autoconf configure scripts is high, the end result is that user's lives are vastly simplified and developers escape the daily barrage of installation questions. Cygnus, of course, benefited too. They had dozens of different machines on which they wanted to support Expect and installing it on each manually was unwieldy and expensive.

Making the switch to Autoconf required a bit of faith to take the plunge, and Tcl dove in as well several months later. This delay didn't directly hurt Expect, but once again it felt like the tail wagging the dog.

In contrast to Autoconf, Tcl caused Expect much worse configuration problems. For instance, Tcl generated a set of compiler flags defining whether certain functions or include files were available. However, these compiler flags were kept private so Expect had to repeat the logic and regenerate the flags during its installation. Only in recent versions, did Tcl finally make available its configuration options in a public way – through tclConfig.sh. Unfortunately, although public, these interfaces remain undocumented.

Problems Successfully Avoided

In contrast to many of the reinvented wheels, Expect also avoided many. Whether this was wisdom or luck is an open question. For instance, early versions of Tcl had no file support. This wasn't a major problem for Expect since it was possible to use existing utilities to read and write files – just as a user would. Indeed, an early paper demonstrated how to retrieve some information out of /etc/printcap by reading it – using ed [Libes 91]! (To be fair, ed did some scanning to locate the information that would have been harder without it or some other similar tool.)

More recent problems revolve around shared/dynamic library support. I steadfastly refused to add shared library support to Expect until the Tcl support had been considerably shaken out – this took several years. Even then, problems remain. For example, Tk's configure does not distinguish between the libraries required for Tk versus those required purely for Tcl. Because some systems object to repeated library specifications when building dependency lists for new shared libraries, Expect has to figure out which libraries it needs separately for Tcl and for Tk. Problems like this would have been evident had Tk been delivered with other shared libraries. In fact, there are many parts of Tcl that could be delivered as separate libraries – socket and channel support are two examples. The reluctance of the Tcl developers to use their own library/package management system has allowed that section of Tcl to be weaker than it would have otherwise been.

A different kind of problem is illustrated by Tcl's idea of main. Originally, Tcl had no main. There was no need for it – Tcl was simply a library after all. But people wanted to write pure Tcl programs – perhaps with the tiniest of extensions. Since this was so prevalent, a main was added to the library. The drawback, of course, was that although Tcl's main satisfied many people, there was always something it didn't quite do – or do correctly. It changed frequently and became an annoyance to extension writers who wanted to rely on it. In contrast, Expect just used its own main, avoiding the problem entirely.

Problems Successfully Fallen Into

Earlier I mentioned that Tcl began providing a regular expression engine (mid '91). To Expect users, it seemed as if Expect simply leveraged that. In fact, Expect's pattern matching needs are not met by the regexp engine. For example, the interact command must be able to report not only 'match' and 'no match' but also whether a pattern *could* match if more characters arrive. Several other pattern matching capabilities are required by Expect and for this reason, Expect includes a complete reimplementation of Tcl's regexp engine as well as Tcl's glob pattern matcher.

The Tcl debugger shows a similar example of this phenomenon. Its ability to allow users to move up and down the stack is implemented by providing a complete reimplementation of TclGetFrame.

Fortunately these types of code rewrites didn't interfere with Tcl itself although they often required dependencies on TclInt.h as well as Tcl's source. The obvious risk here is that they could break at any time and with no recourse.

The I/O driver mechanism in Tcl demonstrates yet another slant on this problem. Expect requires more sophisticated buffering than Tcl offers. Thus, Expect had to provide its own file/buffer system. Expect doesn't use any of Tcl's I/O commands (e.g., read, gets, puts) and Tcl's buffering only serves to slow down Expect's I/O.

Documentation

I considered documentation crucial to Expect. The whole point of the software was so that people would not have to reinvent the automation wheel – and good documentation was a natural extension of that idea – people shouldn't have to figure out Expect either. And although the basics of Expect were intuitive (but then I *would* think so), many novel applications and aspects of Expect were quite non-intuitive and really needed examples and explanations.

The Expect book was a significant investment in my time – about three years from start to final publication [Libes95]. Official NIST funding was not available so I took it on as a personal activity. Much of the first year was spent waiting for permission from NIST and its parent organization, the US Department of Commerce. During that time, it wasn't even obvious that they would allow me to write such a book. But I desperately felt one was necessary. I was inundated by questions and requests for more information, despite having written almost a dozen technical papers for various journals and conferences. (Oddly, this is my first Expect paper at the Tcl/Tk workshop.) I also thought the book would bring a sense of closure to the work – a good theory even if it wasn't true – while at the same time making a contribution to a relatively new and significant field, interaction automation, that hadn't been formally recognized with any other books.

The book also had some other nice effects worth mentioning because they aren't at all evident from reading it. First, writing down an explanation often forces an author to reexamine implementation decisions. There were several aspects of Expect that I rewrote after I realized that my explanations in the book drafts were overly complex for no good reason. These rewrites delayed the book by many months but I was happy each time even though it meant rewriting both code and text in several chapters. The delays in the book were also fortuitous in that Tcl itself was changing. Although Tcl has continued to evolve, the bulk of the book is still accurate. In fact, the only piece of the book that should now be ignored is the section in the last chapter on Expect's timestamp command – this has been superseded by Tcl's clock command. (Expect continues to support timestamp for backward compatibility but the command is officially deprecated.)

The book had other consequences. For instance, it required me to learn more about Tcl. I felt obliged to be accurate about what I wrote and not gloss over things just because I didn't understand them. I also included a tutorial on Tcl itself. Not only were there no good ones at the time, but I still felt that Expect might be considered by many as a stand-alone application for people with no knowledge of (or interest in) Tcl. At the same time, I thought it was important to Tcl itself that it should have a book not written by the author of Tcl, to show that it was a language usable by and significant to others, and to show a set of practical applications.

Writing quality code and man pages is only half the battle. I encourage extension writers as well as application programmers to document their work through web pages and other online documentation, classes, papers, and books. We desperately need more.

Surprises

Many of my experiences with Tcl involved surprises. For instance, I was surprised that Tcl caught on – in the sense that people would write papers and have extended philosophical debates about it. When I began using it for Expect, it seemed irrelevant that no one had used Tcl – or might never use it for any other reason. Tcl looked like such a natural fit that I assumed people would not require any big effort to use Expect. At first, most of the scripts people wrote were just a dozen or so lines. It's hard to imagine how any language could further simplify statements such as:

```
expect "Login: "
send "$username\r"
```

In comparison, Expect rewrites in other languages such as Perl and C are much more clumsy. Although other languages have their advantages for various tasks, Expect is one of the few tools that hasn't been replicated as cleanly elsewhere. I've attempted or assisted such work with six different languages.

Due to constant request, I subsetted the Expect core into a Tcl-less library that could be integrated into other languages such as C and C++. I wasn't happy doing that – I've never written an Expect program that uses C for control. What's the point? Expect itself is neither memory- nor CPU-bound. Most of the time is spent waiting for the spawned program to respond. So Expect doesn't buy much in a compiled language. Interpretable languages like Python and Lisp make much more sense for high-level control tasks [Rossum][Mayer]. Indeed, the Expect library has been successfully integrated with both of these languages.

My philosophies diverged with Tcl in many other ways. For instance, as I had been trained, my early Expect code carefully checked for memory allocation failures. I kept that up well after finding out about Ousterhout's *memory is endless and if it's not, we're all in trouble* policy. Perhaps a year later, I finally ripped it all out. This left my code much more readable yet I felt quite uneasy for some time after. (The standalone Expect library continues to do memory allocation checking since the library can be embedded in other languages and systems that don't share the same policy.)

I was also surprised by users. People put Expect to use in ways I could never have imagined myself, making my own claims for Expect's application seem downright pedestrian. On the other hand, I was occasionally stunned by mail from Tcl users telling me that they couldn't use Expect because it was an extension and their management didn't allow them to use extensions with Tcl.

Change is Hell

Being on the leading edge is painful – change never stops. While Expect is no longer on the leading edge, it is definitely still subject to change. There is still a list of requests for 'improvements'. And a lot of these are understandable – such as porting it to the Windows and Macintosh platforms.

Other changes demand near instant response. Major changes to Tcl can mean major work for me. Sometimes even the most minor changes can mean major work for extension writers. For example, new instances of tclConfig.sh often require substantial study of and revision to Expect's configuration suite.

I have great sympathy for extensions that required changes to the core, such as iTcl [McLennan]. I successfully avoided that trap, although in the case of the debugger, only by sacrificing significant functionality – leaving users without access to line numbers.

Almost as bad however is Expect's use of Tcl's private undocumented interfaces. Such interfaces were often the only way to solve a problem. And since they rarely changed, their impact was insignificant. In fact, the major difficulty was simply listening to people complain about how TclInt.h wasn't available thereby preventing successful compilation of Expect. Distributing TclInt.h with Expect was too risky as it is full of magic numbers that change capriciously.

Lest my observations appear to belittle the Tcl developers, I will say it outright: It is clear to me that they take seriously the changes that impact extensions. Considering Tcl's recent capabilities and improvements, the Tcl developers have done a good job of minimizing the impact.

Clearly, change is a question that weighs heavily on all our minds. When is it acceptable to make incompatible changes? When Cygnus told me that they had Expect suites which performed over a million tests, I realized that there were likely others in similar position and that further changes could no longer be done so capriciously as had occurred in the past. This turned out to be just as well since publication of the book was yet another clear reason not to make any more changes.

Despite the relative stability of Expect's user interface for the past several years, it is likely that Expect will change, partly in order to incorporate ports to the Windows and Mac platforms. It will be very tempting to also try and correct the remaining deficiencies in Expect's user interface at the same time. Even ignoring corrections, Tcl's "reinventions" (especially I/O channels, binary I/O, and multithreading) can only be fully leveraged by redesigning Expect throughout – both the internals and user interface must change. The unnerving aspect is not the amount of work this will take, but the likelihood that future Tcl innovations may require yet more changes.

Concluding Notes

A project, not funded nor planned for the seven year effort it took, necessarily has many chapters to tell and some of them make sense only long after their occurrence. There are many more stories to Tcl – I'm limited here to present only a few. So I've selected stories that illustrate successes as well as stories that illustrate failures. We need both to learn from. And I implore educators who cite this paper not to pull lessons out of context.

Perhaps the most significant lesson of this paper is that what seems like a stable and dependable extension has required a tremendous amount of effort just to keep it working. Not only should the Tcl developers and the vendors appreciate the difficulties they cause but so should extension writers.

Expect originally started out as a small tool. It is still possible to use it that way. But Expect now features over 40 commands and requires a 600-page book to completely describe it and its applications. Like Tcl, Expect has grown. However, I often feel that control is out of my hands. Many of the changes in Expect have been forced by changes to Tcl. While in some cases, a change in Tcl has acted as an enabler, other times it has merely been a stumbling block. Expect's use of Tcl is especially worth critical examination because so many of the underlying assumptions originally motivating the choice of Tcl have changed. On the other hand, Expect's purposes have also changed, due in large part to Tcl's expanded capabilities. While the cost in tracking Tcl has been high, for the most part, Expect has benefited from it proportionally.

Acknowledgments

Thanks to George Howlett, Michael McLennan, David Beazley, Josh Lubell, Przemek Klosowski, Kari Harper, Howard Bloom, Rick Jackson, Chris Kuyatt, and Steve

Ray for reviewing this paper and making suggestions which dramatically improved it.

References

[Dalrymple] Dalrymple, Michael J., "User's Guide To NCFTP And The FTP Protocol", Colorado Center For Astrodynamics Research, University of Colorado, Boulder, CO, undated.

[Diekhans] Diekhans, Mark and Lehenbauer, Karl, "TclX – Extended Tcl", http://www.neosoft.com/tcl/TclX.html.

[Friedl] Friedl, Jeffrey E.F., "Mastering Regular Expressions", O'Reilly and Associates, January 1997.

[Howlett] Howlett, George A., "BLT", http://www.tcltk.com/blt/index.html.

[Libes91] Libes, Don, "Expect: Scripts for Controlling Interactive Processes," Computing Systems, Vol. 4, No. 2, University of California Press Journals, Spring 1991.

[Libes93] Libes, Don, "A Debugger for Tcl Applications," Proceedings of the 1993 Tcl/Tk Workshop, Berkeley, CA, June 10-11, 1993.

[Libes95] Libes, Don, "Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs", O'Reilly and Associates, January 1995.

[Libes97] Libes, Don, "Managing Tcl's Namespaces Collaboratively", Proceedings of the 1997 Tcl/Tk Workshop, Boston, MA, July 14-17, 1997.

[Mayer] Mayer, Neils P., "The OSF/Motif Widget Interpreter", http://www.eit.com/software/winterp/doc/winterp.doc, July 24, 1994.

[McLennan] McLennan, Michael, "[incr tcl] – Object-Oriented Programming in Tcl, Proceedings of the 1993 Tcl/Tk Workshop, Berkeley, CA, June 10-11, 1993.

[Nowitz] Nowitz, D. A., "UUCP Implementation Description", UNIX Programmer's Manual, Section 2, AT&T Bell Laboratories.

[Ousterhout] Ousterhout, John. K., "Tcl and the Tk Toolkit", Addison-Wesley, 1994.

[Phelps] Phelps, Thomas A., "Two Years with TkMan: Lessons and Innovations, Tcl/Tk Workshop, Toronto, Canada, July 6-8, 1995.

[Poindexter] Poindexter, Tom, "Oratcl", Tcl/Tk Extensions, ed., Mark Harrison, O'Reilly & Associates, to appear.

[POSIX] International Organization for Standardization, International Electrotechnical Commission, "ISO/IEC 9945, Portable Operating System Interface (POSIX)", IEEE, New York, NY, 1990.

[Rossum] Rossum, Guido v., Python Language Home Page, http://www.python.org.

[Stein] Stein, L., "CGI.pm: A Perl Module for Creating Dynamic HTML Documents with CGI Scripts", SANS 96, May '96.

Simple Multilingual Support for Tcl

Henry Spencer

SP Systems

henry@zoo.toronto.edu

ABSTRACT

The first, and often largest, step toward internationalization of an interactive application is translation of its output messages, prompts, etc. into the desired language. This is typically done with a "message catalog": the program supplies a key of some kind, which is looked up in the catalog to yield the appropriate translation. While this basic approach is impossible to avoid (given the infeasibility of automated translation), the engineering details matter a great deal.

Programmers can cope with almost infinitely messy interfaces if they must (as witness X programming before Tk), but convincing them to use a new interface voluntarily requires ease of use. Ideally, doing it right should be easier than doing it wrong. If there are concrete benefits from doing it right, doing it right can be *slightly* harder than doing it wrong. However, message-catalog facilities are almost always quite a bit harder to use than *puts*. Hence applications typically use message catalogs only when it is explicitly demanded by the requirements, and retrofitting an application to use a message catalog is painful.

A *carefully designed* message-catalog facility can be almost as easy to use as *puts*, and retrofitting it into a program can be relatively easy. The lookup key should be the message in the original language. Simple provisions for nested subtranslations, untranslated substrings, disambiguating tags, and either explicit (call a procedure) or implicit (done as part of output) translation are essential. Doing all this for Tcl takes care but is practical.

Introduction

Building applications that are portable across computers is easy. Building applications that are portable across national, linguistic, and cultural boundaries is not easy. Many issues need to be addressed: character sets, screen layout, language, formatting conventions, etc. Parameterizing an application to automatically adapt itself to all of these things is an unsolved problem, but much can (and should) be done.

The first, most obvious, and arguably worst problem is language. Most applications have words evident everywhere in the user interface, even when the interface is graphical. Even applications which strive for a purely pictorial interface—not as easy as it sounds, not as culturally neutral as one might think, and often not a good idea—typically have words hidden in places like error messages. Other areas of internationalization can cause misunderstandings and problems, but if users simply can't understand what the application is saying, they can't even get that far.

Automatic translation by software is not yet feasible; human languages are too complex and there is too much context required. Translation of an application's messages, prompts, etc. from one language to another is necessarily done by humans. The most obvious technique, simply modifying the program by editing in a new version of the messages, is unappealing because it creates a new version of the program which must then be maintained separately. What is wanted is a level of abstraction which permits the same program to use different sets of messages depending on the user.

The obvious way to do this, found in many existing applications and even some standards, is a "message catalog": the program supplies a key of some kind, which is looked up in the catalog to yield the appropriate text. (Whether the catalog contains all versions of the text, with an identifier for the language used as a secondary key, or whether a change of language implies switching to a new catalog, is an issue of terminology rather than basic concept.)

While this basic approach is impossible to avoid (given the infeasibility of automated translation), the engineering details matter a great deal. This became clear when a customer (Cancom Business Networks) asked about the possibility of a French-language version of an existing interactive user interface, which (like so many) had been written in English without any consideration of support for other languages. Some preliminary experiments suggested that careful design of the internal interfaces would make a very large difference in the amount of pain involved.

Upon reflection, the problem generalizes, because a facility which is easy to retrofit is also usually easy to use in the first place. The reverse is less often true, because structural decisions are easier to get right the first time than to fix later, but there is still a correlation: facilities which don't have structural implications are much easier to retrofit and are also somewhat easier to use.

The overall desire was for a facility which would be both easy to retrofit into the existing application and easy to use for future applications. We dubbed the resulting software package "Transit".

Interface Design

Programmers can cope with almost infinitely messy interfaces if they must (as witness X programming before Tk), but simple and easy-to-use interfaces are popular for more reasons than just because programmers are lazy. Programming well is hard enough without adding unnecessary difficulties. As a result, there is great temptation to class almost any difficulty as unnecessary unless there is convincing evidence otherwise.

If you want people to do things right, doing it right should be easier than doing it wrong. If there are concrete benefits from doing it right, doing it right can be *slightly* harder than doing it wrong: most good programmers are willing to take a *little* bit of extra effort for well-defined benefits. But a facility which is painful to use will be used only when it is compulsory.

Message-catalog facilities are almost always painful to use. In C, the usual approach is to make the programmer call a function which returns a string to be used as the format for *printf*. Often the messages must be numbered, because the key used to locate the message is numeric. The possibility that items to be substituted into the string by *printf* might not be in the same order in all languages results in clumsy circumlocutions, notably the XPG3 [1] position specifiers found in Tcl's *format* command (which indicate, by number, which of *printf*'s arguments should be converted by each *printf* conversion specifier).

The obvious conversion of this approach into Tcl isn't any happier. In Tcl, one would at least use a string as the key, but other complications intervene.

Tcl does have a *printf* equivalent, *format*, but it is not frequently used for simple output: Tcl substitutions are a simpler and easier way to get things into strings. Precisely because string manipulation is rather easier in Tcl, Tcl output strings tend to be constructed in more complex ways, often by assembling output a piece at a time rather than relying on a single invocation of *format* to do all the work. Forcing Tcl output back into the C mold would be a significant backward step in ease of use. (Running into this problem in a retrofitting experiment was what triggered deeper thought about the problem.)

Moreover, precisely because Tcl is much more string-oriented, the basic structure of Tcl output is often more complicated. For example, an error message reporting a system error during a file access would typically include both the file name and an indication of the nature of the error. In Tcl, the latter arrives in the application as a string, and would itself need translation (given that both circumstances and good judgement forbid meddling with the Tcl core). This means that building a typical output string can involve multiple calls to a translation facility. Things get even worse when dealing with input abstractions, which can build complex messages out of prompts, current values, defaults, etc.

After considerable thought, Transit evolved as a rather un-C-like facility which is both easier to use and more versatile than the typical C message catalog. We think the result is almost as easy to use as *puts*. Some details are still evolving.

Limitations

Certain issues were ruled to be beyond the scope of Transit, to keep it manageable. We believe that solutions for most of these belong elsewhere anyway.

We decided that we would not get involved in the complications of character sets. Our definition of a "character" is whatever Tcl gives us, and a "string" is just a sequence of those. Transit maps strings to other strings. It ignores questions like how many bits are in a character, and whether it takes one or several of them to put a single symbol on the screen. We assume that the message-catalog writer has adequate tools for composing the catalog, and that if Transit pumps the specified strings out, the results will be as desired. Any special arrangements needed to render the characters of the translated message on the user's screen are handled by other means. (We suspect we will probably end up providing for language-specific setup and teardown strings, at the very least, but the issue hasn't come up yet.)

We assume that no serious changes in the technique of interaction are needed. In particular, Transit ignores the issues of character sets whose text runs right-to-left or vertically. Much of this comes under the heading of character sets, discussed above, but it's possible that some amendments to interactive techniques might also be in order. We simply don't know enough about this to deal with it.

Transit makes no particular attempt to address details of formatting of structured values, e.g. whether the decimal point is written as a period or a comma. We think this is best dealt with by the facilities that are doing the formatting, e.g. Tcl's *format* primitive, and we haven't had enough need for it so far to plunge into it ourselves.

Finally, at the moment we're restricting Transit's environment to Tcl, because that's what our current applications are using. (Variable user environments and slow communications links dictated a text-only lowest-common-denominator interface for this work.) Generalizing this to include Tk is certainly interesting, and we've tried to keep an eye on the possibility, but what we have to talk about now was done for Tcl.

The Programming Interface

Our first decision was that the main programming interface to Transit would be named *puts*. With some trepidation, we decided that Transit would rename the real *puts* and define its own procedure by that name, functioning as a wrapper around the real one. While this did incur some worries, it also eliminated a lot of fiddly little code changes in the retrofitting process. The wrapper does translation of any text sent to *stdin*, *stdout* or *stderr*, and leaves everything else alone.[1] This avoids interference with file and network I/O that happens to use *puts*.

The second decision, fairly obvious both from the nature of Tcl and from the choice of *puts* as the interface, was that the key used to look up a message in the Transit catalog would simply be the message itself, in the original language of the program's author. Again, this makes retrofitting tremendously easier, but it also has less obvious benefits. It eliminates the need to define a whole new key space, and also the error-prone maintenance of the relationship between that key space and the program. It provides a "message of last resort" for use when the key lookup fails, instead of having to say "the program wanted to tell you something but I don't know what it was"; even a message in the wrong language is often better than a mysterious number or no information at all. And last but not least, it makes the program easier to read, to write, and to test.

These two decisions, together, meant that a lot of the code didn't have to change at all. Disregarding issues of initialization, the Transit equivalent of

```
    puts "hello, world"
```

is

```
    puts "hello, world"
```

As mentioned earlier, there is a problem with messages which are built up out of multiple parts, by substituting substrings into a master string. Some of the parts themselves need to be translated (e.g., error messages) while some shouldn't be (e.g., filenames). Moreover, the translation of the master string needs to be done *before* substitutions are made, because the contents of the substitutions are unpredictable, so only the pre-substitution master string can appear in the message catalog.

The obvious approach—explicitly invoking translation for each translated substring, and then substituting them into a translated master string—is obviously a nuisance, but worse, it doesn't work very well. Because the order of substitutions may change with translation, the substitution really has to be done with a call to *format*, not with Tcl substitutions. This would require substantial rewriting of retrofitted code and would be a considerable nuisance in new code.

Some thought about the problem led to a crucial observation: the translation of the master string doesn't *have* to be done before the substitution, if the substitution is made reversible. The solution that evolved was to mark substrings within the overall string, and parse the markings at translation time. This typically makes it possible to retrofit translation by just inserting suitable markings, which requires no structural changes to the code. Two flavors of substring brackets are needed, one for translated substrings (henceforth usually just "substrings") and one for untranslated substrings ("literals").

Much of our existing code uses balanced single quotes to delimit things like filenames in error messages. This made it easy to decide how to bracket literals: using balanced single quotes (` ´) for literals greatly reduced rewriting. So this:

[1] It would be desirable to provide for user selection of translated file descriptors, eventually, but so far the need has not arisen. We included *stdin* out of general paranoia.

```
        puts "cannot find file `$name´"
```

results in a message-catalog lookup of "`cannot find file `□´`" (where the □ is used to show the substitution point) and then substitution of the untranslated `$name` into the translation. This does not deal with all cases, but we'll come back to that in a moment.

Translated substrings do need new brackets, especially since *these* brackets must vanish before final output. They have to be reasonably easy to type, because in either retrofitting or new code, there are going to be quite a few of them. We settled on double angle brackets (<< >>) as a distinctive and easy-to-type set of brackets which seldom appear in strings. So this:

```
        puts "*** oops: <<$complaint>>"
```

results in `$complaint` being translated and then substituted into the translation of "`*** oops: □`".

So far, this seems to address only two out of four possibilities. We have substring brackets which vanish before final output, and literal brackets which don't. There is obviously a requirement for literal brackets which vanish, for interpolation of numbers etc. And it's conceivable that there will be a need for a substring which happens to be surrounded by single quotes.

To avoid intruding further on the vocabulary of strings, we decided to use combinations of our existing brackets. To force translation of a quoted substring, we put substring brackets inside literal brackets (`<< >>´); the substring brackets vanish but the literal brackets don't. To substitute without translation but with no quotes in the output, we use the opposite combination (<<` ´>>), which vanishes completely from the output. (The translate-quoted combination is reasonably intuitive; the unquoted-literal one is less so, but seemed the simplest choice.)

Translated substrings are scanned recursively in case they have their own bracketed substrings. Literals are *not* scanned recursively. The latter seemed right in itself, but on inspection it also gave us a bonus: the vanishing literal brackets can be used to quote most of the other brackets, if sometimes a bit clumsily (e.g., <<`<<´>> puts << into the output). This also yields a way of putting single quotes around a literal string which might contain closing single quotes used as apostrophes (<<``isn´t´´>> puts `` `isn´t´ `` into the output), a problem which otherwise resists simple solution.

Initially, we decided that inability to find a string in the message catalog would simply result in the string being used untranslated, with no error generated. Signalling an error in this case would help debugging, but it would also require that the message catalog contain translations for some pretty vacuous strings, which serve only as containers for translated strings. For example, an error-printing routine contains

```
        puts "*** <<$complaint>>"
```

which would require a translation for "`*** □`" if a string with no translation caused an error. This was a valid point, but after some experience, we reversed the decision, with two flourishes.

First, the message catalog can specify a regular expression indicating which strings are too vacuous to require translation. Second, the application can specify a procedure to be called for non-vacuous strings which don't have translations. If the procedure signals an error (as the default one does), that error aborts the translation. If the procedure instead returns a string, then that string is used as the "translation" of the unknown one. Our major application has such a procedure, which logs the problem and then returns the original string enclosed in conspicuous delimiters.

The translation of a (sub)string is not rescanned, so another way to put awkward sequences into the output is to give them names and translate them (e.g., one might define << as the translation of `lb` in all languages, so <<`lb`>> would put << into the output).

This observation spurred another thought. One disadvantage of using the original message strings as search keys is the possibility of duplicate keys, particularly with short substrings. For example, single-letter abbreviations for commands (e.g. `q` "quit") need to be translated, but can easily be used for different purposes in different places in the program.[2] One can obviously avoid this by using an arbitrary unique string as the key, but this neutralizes most of the advantages of using the original-language message as the key. We decided that a *tag* of the form `#string#`, if found at the beginning of a (sub)string being translated, vanishes before output even if no translation is being done.

With the addition of tagging, it is possible to also do limited input handling using the translation facility, by translating the strings and patterns used for input recognition as well. This doesn't fully solve the problems of multilingual input, but so far it has sufficed for our relatively limited and structured interactions.[3]

[2] We do not address the more difficult problem of collisions between such abbreviations *after* translation; preparation of a translated message catalog is not always easy!

[3] There is a bit of a practical annoyance here because Tcl's *switch* statement, in its usually-preferable form (entire body enclosed in a single set of { }) doesn't do substitutions into its patterns. So far we haven't devised any elegant solution for this.

For these and other purposes, it is useful to have a slightly richer programming interface, with some further primitives that are expected to be less often used. The *translate* procedure does the full translation process on its string argument and returns the result; it takes a `-quote` option which wraps the output in <<\` ´>> to prevent further translation. As a convenience for input handling, the *translating* procedure (name chosen because it contains "in") prepends `#input#` to its string argument and then looks the result up in the message catalog. The procedure *untranslatable* specifies a procedure (possibly with arguments) to be called when an untranslatable string is found.

Finally, there needs to be a way to indicate what translation is desired. While almost unlimited complexity is possible here, we've tentatively opted for a very simple solution: *translateto* activates the whole facility, and it takes an argument to indicate the language, which is used to form the filename of a message catalog. (By convention, following various precedents, language "C" means "no translation"; this is not quite the same as not invoking Transit at all, because the various disappearing substring brackets are still stripped out, as are tags.) An optional second argument indicates the directory where the file can be found; failing that, the directory name is obtained from the environment variable `TRANSIT_DIR`. In the interests of localizing performance impact, *translateto* reads the entire message catalog into memory so that lookups can be done quickly.

Message Catalog

For the message catalog itself, we opted for a very simple form with some hooks for future expansion. It's a text file, with the usual text-file conventions of # introducing a comment line, backslash at the end of a line indicating continuation, and empty lines being insignificant.

The message catalog begins with a *prelude*, terminated by a single line containing only "`---`". The prelude contains declarations, one per line, providing overall control; their syntax is that of Tcl commands. After that, the file is a sequence of translations, one per line.

The prelude was originally just a hook for future expansion, with an eye on character-set issues. Now it does have one type of declaration, `vacuous`, for specifying what strings do not need translation. For example, the declaration

```
    vacuous {[^a-zA-Z_0-9]*}
```

says that any string which doesn't contain anything alphanumeric does not require translation.

Each translation is a key, followed by "`->`" (possibly surrounded by white space), followed by its translation.[4] Within the key, substring locations are marked using the translate brackets (not necessarily implying a translated substring, just a matter of not inventing yet another syntax) and arbitrary names, which can then be used in the translation. For example:

```
        change <<x>> to <<y>>   ->   \
            changez <<x>> à <<y>>
```

This avoids the endless counting of parameters found in schemes (e.g., XPG3) where insertions are numbered rather than named.

Keys and translations can optionally be surrounded by balanced single quotes, to provide for including white space at beginning or end (where it is normally stripped) or having one string or the other include the sequence "`->`".

Experience

Experience with this facility has been rather limited as yet, but so far the decisions seem to be working out reasonably well. The bracket syntax is sometimes clumsy, but overall it's causing much less grief than our earlier experiments with a more C-like model in which everything had to be run through a *format* variant.

Using matched single quotes as the literal brackets saved a lot of rewriting in this code, although that's obviously a function of how one usually composes messages. The normal substring brackets are okay; we haven't used quoted substrings yet. The unquoted-literal brackets (<<\` ´>>) see a fair bit of use and are annoyingly clumsy.

A minor annoyance of the bracketing approach is that the brackets foul up string-width calculations in procedures which try to format output (e.g., showing a long list in as many columns as will fit on the screen). So far it's been practical to deal with this by having such procedures invoke *translate* explicitly, do their formatting, and then use unquoted-literal brackets to suppress further translation.

There is obviously a potential problem arising from using brackets that are not *guaranteed* to be absent in normal text. We had one nasty surprise: the first time an error message like

```
    `$type´ invalid
```

was substituted into substring brackets in a statement

[4] We're considering declaring that a line ending in "`->`" is implicitly continued, which would appear to be helpful.

like:

```
        puts "*** <<$message>>"
```

the result was:

```
        puts "*** <<`$type´ invalid>>"
```

which produced major internal indigestion and a garbled complaint about mismatched brackets. We had to do two things to fix this.

First, we revised Transit's error reporting to minimize garbling. In particular, it was a serious mistake for the error message (which naturally got translated!) to attempt to report that it was looking for ´>>! Producing a somewhat less informative message turned out to be the easiest way out of this one.

Second, after a bit of floundering we decided to strip leading and trailing white space within substring brackets before using the remaining contents as a key. This lets us write the problematic statement as:

```
        puts "*** << $message >>"
```

which evades the problem of the two opening brackets merging. We were a bit concerned that the white-space stripping might cause difficulties, but in practice we've found white space showing up at the ends of translated strings only at the outermost level, e.g. in things like

```
        puts "<<$prompt>>: "
```

and since the trailing space is not within any substring brackets, it is exempt from the stripping. (This required minor revisions to the parsing, which originally started out by wrapping the outermost level in brackets to eliminate treating it as a special case!)

We've seen no other problems with nested brackets, even in an application which often has them several levels deep. Some care has been needed to establish conventions for who supplies brackets and who doesn't, but it is immensely convenient to be able to *make* such choices rather than having them dictated by Transit.

There is an obvious problem with using the original messages as keys: it's difficult to systematically enumerate the key space, to be sure you've supplied all necessary translations (especially after the program changes) and that there are no duplicates which need to be tagged. At the moment, all we've come up with is thorough testing, aided by use of an *untranslatable* procedure which logs unknown strings.

For debugging the translation arrangements themselves, it is sometimes desirable to be able to bypass translation in output. This is the dark side of the convenience of having *puts* do translation. The third or fourth time we ran into this, we added a procedure *untranslated*, which invokes the system *puts* without translation, passing any arguments through.

Conclusion

Careful attention to the engineering of a message-catalog facility is important, particularly in Tcl where strings are everywhere and output is built up in complex and flexible ways. Forcing everything into a C-based model, centered on a *printf* equivalent, is awkward for new programs and requires labor-intensive structural changes to old ones. A more sophisticated approach using substring bracketing works much better.

Acknowledgements

Cancom Business Networks first brought up the question of multilingual support, and has been patient while it was sorted out. Although major parts of Transit were developed independently by the author, Cancom was the first guinea pig for it and some of the development necessarily ended up being done on their time; they have graciously agreed that it can remain freeware.

Although Transit is not part of the Shuse account-administration system [2], its first major uses have occurred in connection with Shuse developments.

Doug Berry of Cancom supported and encouraged this work. Ozan Yigit made a number of useful comments, and in particular made a suggestion which eventually turned into the *untranslatable* facility. Toby the cat kept me company during late-night work sessions when everybody else had given up and gone to bed.

Availability

Transit is copyrighted but freely redistributable. It's available for anonymous FTP on *ftp.zoo.toronto.edu* as *pub/transit.tar.Z* .

References

[1] The X/OPEN Group, *X/OPEN Portability Guide*, 3rd edition ("XPG3"), 1990.

[2] Henry Spencer, *Shuse: Multi-Host Account Administration*, in Proceedings of the Tenth Systems Administration Conference (LISA '96), Usenix Association 1996.

Assertions for the Tcl Language

Jonathan E. Cook

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
jcook@cs.nmsu.edu
http://www.cs.nmsu.edu/~jcook

Abstract

Assertions, even as simple as the C **assert** *macro, offer important self-checking properties to programs, and improve the robustness of software when they are used. This paper describes* ASSERTCL*, an assertion package for the Tcl programming language. Our assertions take the form of commands in the program text, and cover point assertions about the computation state, assertions about procedure input values and the return value, and assertions about the values that variables may take on over their whole lifetime. In addition, universal and existential quantifiers are provided for both lists and arrays, not only for individual elements, but for sequences of elements as well.*

1 Introduction

Assertions—declarative annotations to a program that describe some property about the program and its state—are useful tools in producing robust software. Most programming languages are not designed to include assertions, with the notable exception being Eiffel [3], but efforts have produced annotation languages for Ada [2], C [5], Awk [1], and others. This paper describes assertions for the Tcl programming language [4].

Tcl is a popular, interpreted, "scripting" language. It is now being used to build large (tens and hundreds of KLOCs) systems, many of which are being used in the commercial sector. Unfortunately, few tools exist that help one build robust Tcl applications.

This is where assertions fit in. Assertions help developers write robust code by offering dynamic checking of program properties, and enhancing the program's testability. Our ASSERTCL package provides assertions for the Tcl programming language. The assertions take the form of commands in the program text, and cover point assertions about the computation state, assertions about procedure input values and the return value, and assertions about the values that variables may take on over their whole lifetime, singly or in relation to other variables.

In addition, many desirable assertions in Tcl will be over an aggregate data structure, which in Tcl is a list or array. For this, we provide universal and existential quantifiers for both lists and arrays, not only for individual elements but for sequences of elements as well.

Section 2 describes our assertion commands and their meaning and use, and Section 3 describes the quantifiers and their use. Section 4 describes the interface for controlling the evaluation of assertions. Section 5 presents some pragmatic issues in adding assertions to the Tcl language, and describes the methods we used to implement assertions. Section 6 presents some examples in using the assertions and quantifiers. Section 7 evaluates the performance of a Tcl program that uses assertions. Finally, Section 8 concludes with some observations about assertions, the Tcl language and its future, and possible enhancements to the language to make it easier to develop debugging and analysis tools.

2 Assertion Commands

Our assertions for Tcl are commands that use a normal Tcl expression as an assertion about some portion of the program. The four assertion commands that we have added to Tcl are:

- *assert* is an assertion about the current state of computation. It is evaluated at the point it occurs in the source;
- *assume* is an assertion about the input values to a procedure. It is evaluated upon entry to a procedure;
- *assure* is an assertion about the output and return values of a procedure. It is evaluated upon exit from a procedure;
- *always* is an assertion about part of the state space (i.e., variables) of the program. It must always hold, and is evaluated each time one of the variables it depends on changes value.

These commands are summarized in Table 1.

The form for each of these assertions is:

General Form of Assertions	
assert-cmd expr ?fail-action? ?-default_also?	Each assertion evaluates *expr* at some point or points in the execution of the program. If the expression evaluates to true (nonzero), no action is taken. If false (0), *fail-action* is taken if it is specified, otherwise a general exception occurs. The *fail-action* can use break, continue, or return. Specifying the *-default_also* flag will force the general exception to occur after doing the *fail-action*.
Specific Assertion Commands	
assert *expr ?fail-action? ?-default_also?*	Point assertion: evaluates *expr* at the point of its specification, each time the program reaches that point.
assume *expr ?fail-action? ?-default_also?*	Procedure entry assertion: evaluates *expr* at the beginning of a procedure, and should be an assertion about the input parameters of the procedure (and global variables used). This command should be placed at the top of the procedure body, just after any *global* statements, so that it can be seen as part of the procedure specification.
assure *expr ?fail-action? ?-default_also?*	Procedure exit assertion: evaluates *expr* just before returning from a procedure, and should be an assertion about the return value of a procedure, and any side effects (global variables modified) of the procedure. Note that this command does not need to be placed at the end of the procedure, and should be placed at the top, after any *assumes*, so it can be seen as part of the procedure specification. The values of the input parameters at the time of procedure invocation are available through **_in* variables, one for each parameter. The return value of the procedure is available through the *return_val* variable.
always *varlist expr ?fail-action? ?-default_also?*	Program state assertion: evaluates *expr* every time one of the variables in *varlist* changes. The variables in *varlist* should be (a subset of) the variables in *expr*, and the expression should be a statement about the relationships that must hold or values those variables can take on. The assertion exists for the lifetime of the variables in *varlist*, and the *always* assertion can be used in procedure bodies for local variables, as well as for global variables.

Table 1: The four basic assertion commands for Tcl.

assert-cmd expression [failure-action] [-default_also]

where *assert-cmd* is one of *assert*, *assume*, or *assure*. The *always* assertion is:

always varlist expression [failure-action] [-default_also]

In both forms, *expression* is the Tcl expression to be evaluated. For the assertion to be satisfied, this expression must evaluate to true (nonzero). For the *always* assertion, *varlist* is a list of variables to activate the assertion on, such that when any one of them changes, the expression is evaluated. It does not have to include all the variables in the *always*,[1] but should not include any variables not in the expression.

Failure-action is an optional Tcl statement (block) that, if supplied, will be executed if the assertion fails. If no *failure-action* is specified, the default action of producing a general exception occurs, printing out the nature of the failed assertion and aborting the program. *-default_also* is an optional flag that, if specified, will execute the default action after executing the *failure-action*; this can be used to print out detailed messages in the *failure-action*, while still aborting the program.

[1]For example, there may be transient states for one of the variables that should be ignored.

For assertions about (and in) procedures, the value of the input parameters at the time of invocation is useful, because the parameters may be changed during execution of the procedure, and an assertion would no longer have access to the original values. For this, we create variables of the form *param_in*, where *param* is the name of each parameter to the procedure. These **_in* variables are set with the input value of the parameter, and do not change over the execution of the procedure. They can be used in any assertion inside that procedure.

For *assure* assertions, the return value of the procedure must also be available. We provide this through a variable *return_val*. This variable is set in evaluating an *assure*, when the procedure is exiting. While it can potentially clash with an existing variable name (as can the **_in* variables), if the variable is local, no harmful effects will occur since the procedure is exiting, though the original value will not be accessible in the *assure*. The only case where a problem will occur is if *return_val* is a global variable and is declared so in the procedure.

General Form of Quantifiers	
quantifier-cmd item-varname[s] data-struct expr	A quantifier command evaluates *expr* for all items or item sequences in the data structure (list or array), and returns 1 if the expression is true (nonzero) for the specified quantification over the data structure. If *item-varname* is singular, that variable takes on the value of each item in the list (or index in the array) as the quantified expression is evaluated. If *item-varname* is a list, the variable names in the list take on successive values in the data structure, such that the order of names in the parameter is the order of values in the data structure. If more *item-varname*'s are specified than there are members in the data structure, the quantifier returns 1.
Specific Quantifier Commands	
lall *item[s] list expr*	List universal: evaluates *expr* for all items or item sequences, and returns 1 if the expression is true over the whole list, and 0 otherwise.
lexists *item[s] list expr*	List existential: evaluates *expr* for all items or item sequences, and returns 1 if the expression is true for any item (item sequence) in the list, and 0 otherwise.
rall *index[es] array-name expr*	Array universal: evaluates *expr* for all indices or index sequences, and returns 1 if the expression is true over the whole array, and 0 otherwise. The index list can be prepended with any sorting flags that 'lsort' accepts.
rexists *index[es] array-name expr*	Array existential: evaluates *expr* for all indices or index sequences, and returns 1 if the expression is true for any index (index sequence) in the array, and 0 otherwise. The index list can be prepended with any sorting flags that 'lsort' accepts.

Table 2: The four quantifier commands for Tcl.

For this case, we would strongly suggest that *return_val* is not a very good name for a global variable.[2]

2.1 Usage Conventions

Assertions in a program become part of the documentation of the program, its behavior, and its internal interfaces between modules and procedures. This suggests that some of the assertion forms should have standard placements in the program.

The *assert* assertion is a point assertion. It should be used at any point in the Tcl program where the programmer can make a succinct declarative statement about the state of the program, or where they want to ensure that a variable or variables contain the proper data. At the top or bottom of a loop body it can act as a loop invariant.

For procedure interfaces, the *assume* and *assure* assertions should be used. These assertions capture the assumptions that the procedure makes on its parameters and any global variables used, and the assurances of its return values or of any global variables changed. The assertions in effect become part of the declaration of the procedure interface. As such, they should appear as the first statements in the procedure body, after any *global* statements (which are also part of the interface).

The *always* assertion, when it is used as an assertion about global state, should appear at the top of the given Tcl module that has the global state variables. If it is being used as an assertion about the internal state of a procedure, it should appear at the top of the procedure body, after any *assume* and *assure* assertions, or after the variables are created.

[2]There are safer ways for naming input parameter and return value variables, such as using an array with named elements, like *assert(return)* for the return value. It was felt that a construct such as this would simply be too cluttering for being able to succinctly read an assertion expression.

3 Quantifier Commands

The two basic aggregate data types in Tcl are the list and array. Assertions may often take the form of describing the properties of a list or an array; for example, specifying that a list contains all positive numeric items. In these cases, universal and existential quantifiers over lists and arrays are useful. Our commands for these take the general form of:

$$quantifier\ \{item|index\ list\}\ \{array\text{-}name|list\}\ expression$$

Our four quantifiers are *lall* and *lexists* for list universal and existential quantification, and *rall* and *rexists* for array quantification. These commands are summarized in Table 2. The first parameter to a quantifier is a list of item names or index names (dependent on whether a list

Assertion Control Interface	
assertcl *disable ?all|proc|var? ?all|proclist|varlist?*	
	Disables all assertion checking, assertion checking for the given list of procs, or assertion checking for the given list of variables. The keyword *all* can be used in place of a procedure or variable list, indicating all procedures or all variables, respectively.
assertcl *enable ?all|proc|var? ?all|proclist|varlist?*	
	Enables all assertion checking, assertion checking for the given list of procs, or assertion checking for the given list of variables. The keyword *all* can be used in place of a procedure or variable list, indicating all procedures or all variables, respectively.

Table 3: The control interface for AsserTcl.

or array is used). These names are variables that take on successive values of the items in a list (indices into an array). The quantifiers have their general meaning: the expressions in *lall* and *rall* must hold for all sequences of items (indices), and the expressions in *lexists* and *rexists* must hold for at least one sequence of items (indices). A quantifier returns 0 for false and 1 for true.

For example, to specify that all elements of a numeric list are positive, the quantifier

```
lall i $list {$i >= 0}
```

is used. Here the item list is only one item, *i*. One item, however, cannot be used for relations among items, such as specifying that a list is sorted. For this, a quantifier such as

```
lall {i1 i2} $list {$i1 <= $i2}
```

is used, where *i1* and *i2* take on successive values of items in the list. On a list of five items, for example, there would be four pairs of items to compare in the quantifier. If a list is shorter than the number of items asked for, the quantifier returns true (1).

Array quantifiers work the same way, except for one problem—the order of indices to an array is unconstrained,[3] so quantifiers over sequences of elements are less obvious. Still, in many uses of arrays, programmers do have a sequence of indices in mind, and it would not be unreasonable to specify that an array is sorted, like

```
rall {i1 i2} Arr {$Arr($i1) <= $Arr($i2)}
```

For this, our quantifiers process the indices in an *lsort*'ed sequence; that is, we use an index sequence returned by *[lsort [array names Arr]]*, in this example.[4] However, *lsort* by default is an ASCII string sort, and if the indices are numeric, this will give the wrong order. For this, our index list can take any flags that *lsort* takes, and we pass these on to *lsort*. Thus, the above quantifier would be changed to

```
rall {-integer -decreasing i1 i2} Arr \
     {$Arr($i1) <= $Arr($i2)}
```

[3]In Tcl, arrays are associative, and any string can be an index into an array.

[4]Of course, a quantifier using only one index does not call *lsort*, since there is no order needed.

to specify integer indices in a decreasing order. This syntax is a bit cumbersome and unfortunate, but it is the best we can do.

4 Controlling Assertion Checking

We recognize that assertion checking can be costly in some instances, in terms of performance. This can be especially true with quantifiers over large data structures, and the cost can be somewhat hidden at times. For example, if a procedure is called for each item in a data structure, and that procedure has an assertion about the whole data structure, the assertion effectively turns an $O(n)$ computation into one that is $O(n^2)$.

We expect that when a program is finally released for use, disabling the assertion processing is desirable. Even during system development, as some portions of the system become reliable, it may be desirable to turn off assertion processing for those portions.

In AsserTcl, assertions are controlled through the *assertcl* control interface. Table 3 shows the commands that are allowed by this interface.

Assertion control is provided at the procedure level, variable level (for *always*), and the global level. The global level simply disables all assertion processing, and is effected by issuing the command

```
assertcl disable all
```

for disabling, and with the *enable* keyword for enabling.

By using the *proc* keyword and providing a list of procedure names, assertion disabling can be done per procedure. The command

```
assertcl disable proc P1 P2 P3
```

would disable assertion checking for the three named procedures. Other assertion processing would still take place, provided that global assertion checking had not been disabled. The keyword *all* in place of a list of procedure names disables assertion checking for all procedures.

A similar use of the *var* keyword disables processing any *always* assertions for the specified variables; if a given *always* assertion still relies on other enabled variables, however, it will be processed when those variables change. The keyword *all* in place of a list indicates all variables with *always* assertions.

Even with global-level assertion disabling, an overhead of checking the control variables is still paid. Thus, for a situation such as final release of the software, a *nullassertcl* package that declares empty assertion commands is available. Requiring this package in place of the regular *assertcl* package source leaves just the call of an empty procedure as the only overhead.

Section 7 analyzes the run-time performance of these various levels of assertion checking.

For removing virtually all overhead of using assertions, our future work is expected to include making a program processor that will comment and uncomment assertion commands automatically. This will allow virtually no overhead (except for comment-skipping) to exist in a released program, while still allowing the assertions to remain as documentation, and to be reactivated if needed.

5 Implementing Tcl Assertions

The main Tcl command that enables the addition of assertion commands is the *uplevel* command, which evaluates a script in a different context; thus a procedure can be passed an expression or even a block of statements, and it can evaluate those in the context of its caller, thus making the procedure look like a command in its caller's context.

The other main Tcl feature that we use is the ability to rename an internal command and replace it with our own procedure. We implement a *proc* procedure that acts as a front-end processor to the real *proc* command. Our front-end searches the procedure body for assertion commands, and does the following two things if it finds any.

1. The body of the procedure gets prepended with statements that copy parameter values to a corresponding *param_in* variable. This gives access to the input values regardless of whether the procedure modifies the real parameter variables. Only those *param_in* variables that are used are copied.
2. Each *return* statement in the procedure gets replaced with our own *assertReturn* statement. Ours takes care of evaluating the *assure*'s for that procedure, and then returning from the procedure.

After these two operations on the procedure body are done, we then call the normal *proc* command, so that Tcl registers the procedure. The implementation of *assure* is explained in more detail below.

5.1 Assert, Assume, and Always: The Easy Ones

The *assert* command is implemented in the straightforward manner of a procedure that acts like a command, i.e., *uplevel*'ing the expression to be evaluated and catching any returned exceptions. If the expression itself generates an exception (an error or a user-defined exception), *assert* passes this back to the enclosing context. If the assertion fails, *assert* by default generates its own exception using the *return* command. If the *assert* has an associated *failure-action*, however, that action will be *uplevel*'ed rather than generating an exception.

The *assume* command is really just an alias for the *assert* command, implemented as a procedure that *uplevel*'s an *assert* call. This does not quite fit the definition of an *assume* being evaluated at the start of a procedure invocation, since if the *assume* statement is not placed at the beginning of the procedure body (like we suggest), it will not be evaluated at the start. The alternative would be to preprocess the procedure body and move the *assume*'s, but this would involve some serious syntactic analysis. For now, our decision is to forego this step and rely on conventional placement of the *assume* commands.

For implementing the *always* assertion, the Tcl *trace* command is used, which allows registration of a procedure to call each time a variable is written (reads are also traceable). The *always* assertion creates a uniquely named procedure that evaluates the expression in the assertion and processes any exceptional conditions and a *failure-action*, if there is one. This procedure is then attached to each variable specified in the *varlist* parameter using the *trace* command.

5.2 Assure: The Hard One

Implementing the *assure* command is quite a bit different and more involved. For this command, we do need to modify the procedure body. Due to the *return* command being a not-so-general exception generating mechanism, we do not (and cannot) globally replace the *return* command, but we process the procedure's body and replace each *return* in a procedure with our own *assertReturn* procedure, if the procedure being processed has any *assure*'s. The *assertReturn* procedure evaluates the *assure*'s, and then effects a real *return* from the calling procedure.

Replacing the *return* command can cause some problems and incompatibilities with existing Tcl behavior, but only under well-defined circumstances. The problem stems from the fact that *return* is not simply a command that returns a value from a procedure, but rather a mechanism for generating exceptions. The exception generated by a *return* is passed by Tcl to the context of the caller of the procedure that the return command is in. But when we replace the *return* with our own procedure, we add a layer of procedure call, so that now the context that gets the exception is our caller, not our caller's caller, as it should be. And *return* cannot be *uplevel*'ed, because of the way the call stack is processed in Tcl.

Nevertheless, for Tcl code that simply uses *return* in the normal fashion of returning a value from a procedure,[5] replacing the command with our own does not break any Tcl behavior, and allows us to evaluate any *assure*'s and then effect a real return.

The lexical replacement of *return* with *assertReturn* is not done globally or universally. We only do this in the body of a procedure that is using *assures*, so that any other use of return is not affected. Furthermore, we only replace a return call that begins on its own line (white space excluded). This not only offers simple and quick replacement, avoiding complex syntactic processing, but offers an "out" to a programmer who needs to use return to throw an arbitrary exception in a procedure using *assures*—they can hide the return from our processing by using a construct such as

```
if 1 { return -code $Exception ... }
```

Since this return does not begin its own line, our replacement method will ignore it.

The *error* command can also be used to return from a procedure, but since this method is not interceptable (due to the above-mentioned limitations on the *return* command), we make no effort to catch this command. In [4], it is recommended that *error* not be used except for true errors, in any case.

6 Examples

This section presents a few simple examples to give a flavor for what assertions in Tcl look like and can do.

A point assertion about variables x and y:

```
assert {$x>4 && $x<$y+2}
```

A point assertion that simply prints a warning and does not abort the program:

```
assert {$x>4 && $x<$y+2} {
   puts "Assert failed: x:$x y:$y"
}
```

An assertion about some global variables over the life of the program:

```
always NumUsers {$NumUsers >= 1 && \
                 $NumConnected > $NumUsers}
```

Note that this assertion only gets checked when *NumUsers* changes, not when *NumConnected* changes.

A simple procedure declaration with interface assertions:

```
proc square {x} {
   assume {$x+1 > $x}  ;# tests for numeric value
   assure {$return_val == $x_in * $x_in}
   set x [expr $x * $x]
   return $x
}
```

[5] A procedure return "exception" is the only one that can be passed up one extra level.

A procedure that returns the quotient and remainder of a divide operation:

```
#
# divmod: returns a two-element list of quotient
#         and remainder
#
proc divmod {dividend divisor} {
   # simply test for numeric value
   assume {$dividend + 1 > $dividend}
   # also, make sure non-zero divisor
   assume {$divisor + 1 != 1}
   # must return the correct quotient and remainder
   assure {[lindex $return_val 0]*$divisor_in +  \
           [lindex $return_val 1] == $dividend_in}
   # procedure body
   set quot [expr $dividend / $divisor]
   set rem  [expr $dividend % $divisor]
   return [list $quot $rem]
}
```

Assert that all list elements are positive:

```
assert {[lall item $list {$item>=0}]}
```

Assert that a list contains an element "Jon":

```
assert {[lexists item $list {"Jon" == $item}]}
```

Assert that a list is sorted:

```
assert {[lall {i1 i2} $list {$i1 <= $i2}]}
```

Assert that an array contains the value 42:

```
assert {[rexists i Arr {$Arr($i) == 42}]}
```

Assert that an integer-indexed array is sorted:

```
assert {[rall {-integer i1 i2} Arr \
       {$Arr($i1) <= $Arr($i2)}]}
```

Of course, an arbitrary Tcl procedure can be called in an assertion (though it should not have side effects!), so a procedure that checks the consistency of a more complex data structure can be used, like:

```
assert {[lall cust $CustomerList \
       {[CheckCustRecord $cust]} {
   puts "Warning: Customer is malformed -- $cust"
}
```

which asserts that a list of customer records are all consistent, but only prints a warning if it is not.

7 Performance Evaluation

While assertions serve as unambiguous, written-down, declarative specifications of a program behavior, their compelling reason of existence is that they can be evaluated at run-time, providing increased assurance that a program's behavior is correct.

Run-time evaluation of assertions does not come for free, of course. They add extra processing to a program and can add significant increases in program execution time. For example, in the case of an assertion using *lall*, the expression must be evaluated over the whole list to evaluate the assertion. If it is in a loop, each iteration

Assertion Configuration	Tcl 7.6	Tcl 8.0a2
full assertions	97.53	48.16
-P1	79.43	39.32
-P2	78.12	38.64
-P1,P2,P4	62.17	28.51
-P3	35.91	23.63
all disabled	3.69	2.87
-P1-8	2.56	2.29
nullpackage	1.14	0.73
comment-out	0.95	0.66

Table 4: Performance of assertions on Tcl 7.6 and Tcl 8.0a2

of the loop processes the whole list, potentially making an $O(n)$ operation become $O(n^2)$.

This overhead can be controlled during development and testing through the judicious enabling and disabling of assertions in procedures and variables, as explained in Section 4.

Here we present some performance numbers for a small Tcl program developed using assertions. The program was a non-interactive, stochastic Petri net simulator, and made heavy use of arrays. The program consisted of thirteen procedures, all of which had assertions in them. Two procedures used the *lall* quantifier in their assertions. The program did not contain any *always* assertions. The program was 177 source lines, excluding comments and brace-only lines, and had 16 assertions.

We ran the program using both Tcl 7.6 and Tcl 8.0a2, on an UltraSparc running Solaris. Table 4 shows the performance of a Tcl program over various configurations of assertion checking. Each value is the mean of 5 runs, timed using the Unix *time* command. Rows with '-P#' configurations have assertion checking disabled for the specified procedures, but enabled for everything else. The last line in the table was the running time of the program with all assertion statements hand-commented out of the program. This represents the lower bound on running time.

As can be seen, full assertion checking carries a high price for this particular example. It runs almost 100 times slower on Tcl 7.6, and almost 75 times slower on Tcl 8.0. Disabling various procedures reduces the time by considerable amount, with the disabling of assertions in P3 cutting the execution time by over half on Tcl 8.0, and almost two-thirds on Tcl 7.6.

The two methods of disabling assertion checking, using *assertcl disable all* or using the *nullassertcl* package both show good performance relative to the commented-out performance. The *nullassertcl* package performance is within 20% on Tcl 7.6 and within 11% on Tcl 8.0, indicating that it is deployable technology—with an interactive Tk program, an 11% overhead would not be noticeable in most instances.

An interesting note is that disabling assertions in eight of the thirteen procedures results in performance that is faster than the global *assertcl disable all*; this happens because the procedure disabling actually skips inserting any assertion processing in the body of the disabled procedure(s). Thus, if the time-critical procedures are disabled in this manner, the other procedures can still check assertions, and the performance remains good.

For the list quantifiers, allowing the commands to use list names rather than lists would improve their processing time immensely.[6] However, this usage would not be consistent with the normal Tcl list commands, so we chose to leave the calling interface as using a list. However, we may decide to change this, or support both, in the future.

8 Conclusion

In this paper we presented AsserTcl, an assertion package for the Tcl programming language. We feel that assertions will make the development of robust systems easier. At the same time, we hope to use this foundation for future research exploring more ideas in assertions for programming languages, and for evaluating the effectiveness of assertions.

Tcl is an evolving language, and the next version will have *namespaces*, which will be akin to packages or modules. Earlier work on Ada packages [2] and Eiffel objects [3] have shown that modules present interesting issues in developing effective assertions for them. In addition, object oriented extensions to Tcl exist, and these can provide a platform for exploring issues about assertions for classes and objects.

In developing this package, some limitations were encountered that could be ameliorated by adding some functionality to the Tcl core language. Our suggestions are:

1. Extend the exception generating capability of the *return* command so that it is possible to throw an exception to a specified level, rather than always the caller. This would enable the true replacement of the *return* command (enabling other debugging packages), and would generalize the exception mechanism.

2. Extend the *info* command with access to a procedure's current line number (when executing) and file name. This is also a feature that would enable more functional debugging packages. The *info level* command could be extended to provide the line number currently being executed in that level, and a command such as *info proc procName* could be added that would return the file name and the beginning line number of the procedure definition.

[6]A simple test with a 20-element list showed an order of magnitude in speed difference.

Availability

AssertTcl is, and hopefully will remain, a Tcl-only package, to ensure the best portability and ease of installation and use. It is freely available at http://www.cs.colorado.edu/~jcook/TclTk.

Acknowledgements

I would like to thank Professor Mikhail Auguston for discussing aspects of assertions for programming languages, and to Professor David Rosenblum for first turning me on to assertions. Many thanks to the anonymous reviewers of this paper as well. Their suggestions improved it greatly.

REFERENCES

[1] M. Auguston, S. Banerjee, M. Mamnani, G. Nabi, J. Reinfelds, U. Sarkans, and I. Strnad. A Debugger and Assertion Checker for the Awk Programming Language. In *Proc. International Conference on Software Engineering: Education and Practice*, Dunedin, New Zealand, 1996.

[2] David Luckham. *Programming with Specifications: An introduction to Anna, a language for specifying Ada programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1990.

[3] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, New York, 1988.

[4] John K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[5] David S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 1995.

Extending Traces with OAT[1]: an Object Attribute Trace package for Tcl/Tk[2]

Alex Safonov, Joseph A. Konstan, John V. Carlis and Brian Bailey
Department of Computer Science
University of Minnesota
{safonov,konstan,carlis,bailey}@cs.umn.edu

Abstract

Tcl supports variable traces, which associate arbitrary scripts with variable reads, writes and unsets. We developed OAT (Object Attribute Traces), a protocol for extending traces to attributes of arbitrary Tcl "objects." We wrote several OAT-based extensions including TkOAT, which provides traces on attributes of Tk widgets and canvas items. The OAT protocol and derived extensions bring the benefits of more expressive constraints to Tcl/Tk applications by providing extended traces. OAT requires no changes to the Tcl core and is implemented as a loadable library; OAT-based extended trace packages introduce minimal changes to the code of existing extensions (Tk, CMT, etc.). The new version of our formula manager, TclProp, takes advantage of extended traces provided by OAT.

1. Introduction

Tcl's trace mechanism allows the Tcl script programmer to specify arbitrary scripts to be executed when a given variable is read, written, or unset. It provides, among other things, a good means for propagating changes to variables [Sah95]. For instance, Tcl uses traces internally for the following:

- tracking changes to the `tcl_precision` variable, to update the floating-point precision and printing format in the interpreter.
- monitoring the `env` array to propagate changes to the corresponding environment variables.
- supporting linked Tcl and C variables.
- implementing the `vwait` command.

The built-in trace mechanism is limited to variables only. This limitation becomes significant, for instance, when one attempts to use traces and formulas with UI widgets. It is often desirable to detect changes to the state of Tk widgets, or to link widget attributes with formulas. Because many attributes of Tk widgets are not associated with variables, they are not traceable. Examples of such widget attributes include:

- button state, normal or disabled
- state of a menu item, normal or disabled
- button or label color and bitmap
- the number of items in a listbox

We propose OAT, a generic protocol for extending Tcl traces. We target two types of users: Tcl script programmers and Tcl extension developers. Script programmers will benefit from the OAT-based extended traces, because the trace-based Tcl code is more compact and easier to maintain. Extension developers can use OAT to make their objects traceable, to bring the benefits of declarative trace-based programming to users of their extensions. We discuss our experience of making Tk widgets and CMT clocks traceable. Finally, we describe the OAT implementation.

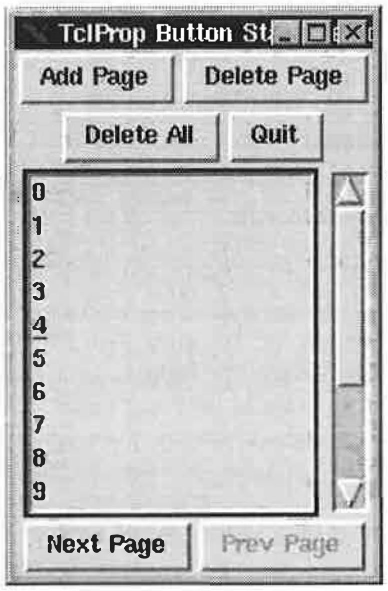
2. Extended traces

The Listbox Pager shown in Figure 1 demonstrates the usefulness of traces set directly on widget attributes. The "Prev Page" and "Next Page" buttons scroll the contents of the listbox by the number of visible lines. They are disabled when the listbox is positioned on the last page (starting at item 5) and the first page (starting at item 0), respectively. The state of the pager buttons depends on the following three attributes of the listbox:

1. the total number of items
2. the number of visible items
3. the number of the first visible item

[1] One of the reviewers suggested that we come up with a different name for the OAT package, so we renamed our package FATCAT: Flexible Architecture for Tcl Constraints And Traces.

[2] Partially supported by NSF grants EBN-9419233 and IRI-9410470, and a grant from the Minnesota Distributed Multimedia Research Center.



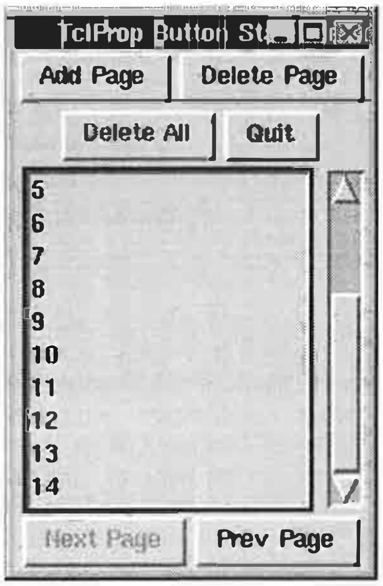


Figure 1. A listbox pager using extended traces: first and last pages

Updates to these three listbox attributes can be scattered throughout the script code and widget callbacks. Without extended traces, these updates must be followed by the code that enables and disables the “Prev Page” and “Next Page” buttons. With traces on the listbox attributes available, the Tcl programmer only needs to specify the code to set the state of the pager button once, in the trace callback. The working code in Figure 2 creates the extended traces on listbox attributes, and defines a procedure to check whether the listbox is on the last page, and to enable or disable the buttons accordingly. The trace callback that controls the state of the “Prev Page” button is similar. Since listbox attributes are not traceable in stock Tk and no variables are associated with them, this example demonstrates the need for extended traces for UI development.

Extended traces are useful for canvas item "geometry management". Suppose we would like to keep a canvas rectangle twice the size of another one as the latter is resized. We create the trace on the `coords` attribute of the source rectangle, and associate with it the code to resize the target rectangle. Similarly, the trace on canvas item `coords` can maintain its width-to-height ratio constant, ensuring, for instance, that a rectangle remains square.

Extended traces on canvas attributes also help to maintain geometric relationships among them. For instance, graph drawing packages [Ellson96]

```
# create extended traces on topIndex, numElements, and fullLines
# attributes of listbox widget .lb
trace widget .lb topIndex    w setNextPageState
trace widget .lb numElements w setNextPageState
trace widget .lb fullLines   w setNextPageState

# define trace callback
proc setNextPageState {nameSpace widgName attrName op} {

    set fraction2 [lindex [$widgName yview] 1]
    if {$fraction2 < 1} {
        .nextPage conf -state normal
    } else {
        .nextPage conf -state disabled
    }
}
```

Figure 2. Code for trace-based “Next Page” button

and programs for structured drawing benefit from the ability to keep canvas items attached as they are dragged. A line is attached to a circle by creating traces on both line and circle coordinates[3]. The code associated with the traces moves one item as the other is dragged.

3. Traceable types and the OAT protocol

These examples and our experience with Tcl/Tk development motivated us to create a protocol for defining traces on attributes of arbitrary Tcl "objects" [Roseman95]. We call an object type enhanced to support traces on its attributes a *traceable type*. Variables constitute a traceable type with only one (implicit) attribute: their value. Other traceable types we created are Tk widgets, Tk canvas items, CMT clocks, and [incr Tcl] namespaced variables and objects (work on the latter is still in progress).

Our primary goals in the development of OAT were:
1. to define a clean protocol for easily adding traces to new and existing extensions that employ the concept of state-holding objects;
2. to extend the benefits of traces to arbitrary Tcl objects.
3. to make OAT and OAT-based extensions easy to integrate with existing Tcl/Tk installations.

As a proof of concept and for general use, we developed TkOAT, supporting traces on Tk widgets and canvas attributes, and MediOAT, adding traces to CMT clocks. We tried to conform as much as possible to the Tcl interface provided by the trace command, and internal C interfaces, since we found these well-designed and implemented. To ease integration of OAT and OAT-based extensions with Tcl/Tk installations, we created the extensions as loadable libraries, and made changes to the original code only where it was absolutely necessary.

The OAT protocol consists of two parts:
- Tcl API, used by the Tcl script programmer. It defines creation, querying, and deletion of extended traces on objects of traceable types from Tcl code.
- C API, used by the Tcl extension developer. The OAT C API defines a protocol for creating new traceable types, and checking and triggering traces on attributes of traceable type objects.

[3] While the coordinates are not, strictly speaking, configurable "attributes" of canvas items, for cases like these it is desirable to treat them as such.

TclProp [Iyengar95], a trace-based formula manager for Tcl, was updated to take advantage of extended traces. The new TclProp API includes type identification for objects whose attributes are terms in a formula.

3.1 OAT Tcl API

The OAT Tcl API permits a programmer to manipulate extended traces from Tcl scripts. The interface is modeled on variable traces. As with variables, each traceable type has three subcommands for the trace command: to create, query, and delete a trace. Code in Figure 2 shows an example of creating traces on attributes of listbox `.lb`; the "`widget`" keyword is the trace creation subcommand for the Tk widget traceable type. The general syntax of the extended trace command, shown in Figure 3, is the same as in stock Tcl. However, the "option" subcommand can be a trace keyword for any of the registered traceable types. The keywords for the subcommands are defined when a traceable type is created, as discussed below.

```
trace option ?arg arg ...?
```

Figure 3: The extended trace command syntax

A Tcl programmer can query OAT for all registered traceable types; this is accomplished with the "`oat`" command. It returns the names of all registered traceable types; the list always includes the built-in traceable type, `variable`. Suppose the TkOAT extension is loaded, providing traces on widget and canvas attributes. The `oat` command will return the following list: "`variable widget citem.`" Here "`widget`" and "`citem`" are additional traceable types registered by TkOAT. The "`oat typename`" command returns the subcommand keywords for a registered traceable type, where "`typename`" is a name of a traceable type. The "`oat widget`" command will return the trace subcommands for traceable type widget: the list "`widget winfo wdelete.`"

3.2 OAT C API

The traceable type creation protocol permits a writer or a maintainer of a Tcl extension to define new traceable types. The creation of a traceable type involves three main steps:
- **extend the trace command**: the trace command is extended to accept keywords for new sub-

commands that create, query, and delete traces on this type.

- **register the callback function**: the OAT-supplied or custom C callback is registered to be called when these keywords are supplied with the trace command.
- **insert checks for traces**: checks for traces are inserted in the attribute update code for the traceable type.

Extend the trace command. A traceable type can be defined by the extension developer at different levels of detail. At a minimum, the name of the new traceable type must be supplied. The OAT library will automatically generate the subcommand keywords for the trace command from the type name. For instance, if "`gadget`" is specified as a traceable type name, the keywords to create, query, and delete traces on it will be "`gadget`", "`gadgetinfo`", and "`gadgetdelete`", respectively. If desired, these three keywords can be explicitly specified. As an example, Tk widgets form a traceable type with trace subcommands "`widget,`" "`winfo,`" and "`wdelete.`". Figure 4 shows the OAT traceable type data structure, along with field names, types, and brief content descriptions.

Register the callback function. The developer can use the standard OAT-supplied callback for trace subcommands, or write a custom one. The first approach significantly simplifies creating a traceable type. The extension developer should use the OAT-supplied callback if all extension objects are "global" like Tk widgets, that is, are not contained within a namespace.

Object systems can require additional information to identify an object being traced. For example, traces on canvas item attributes require both a canvas name and an item tag or id to uniquely identify the traced attribute. To support this behavior, OAT allows the extension developer to specify a custom C function that is called when the Tcl interpreter processes the trace subcommands for this type. The prototype for the OAT custom trace callback is shown on Figure 5. We are investigating how to make OAT-supplied trace callbacks more general so that they accommodate objects with namespaces.

Insert checks for traces. The extension developer needs to insert the check for traces in the C code where extension object attributes are updated. This check is done in a single line of code, as shown on Figure6.

Field Name	Field Type	Comment
`typeName`	`char*`	traceable type name
`traceCreate`	`char*`	keyword for trace command
`traceInfo`	`char*`	keyword for trace command
`traceDelete`	`char*`	keyword for trace command
`traceCmdProc`	`function pointer`	function to call when trace subcommand for this type is supplied, or `NULL` (use default callback function)

Figure 4: A traceable type data structure

```
typedef int (Oat_CmdProc) (ClientData dummy,
                           Tcl_Interp* interp,
                           int argc, char* argv[]);
```

Figure 5. Type definition for the custom OAT trace callback.

```
/* Call traces for widget or canvas item attribute being configured. */
Oat_CallObjTraces(interp, objRec, specPtr->argvName, TCL_TRACE_WRITES);
```

Figure 6. Adding check for extended traces to `tkConfig.c`

```
char* Oat_CallObjTraces (
      Tcl_Interp* interp,   /* current interpreter            */
      char*       objPtr,   /* pointer to object data structure */
      char*       attrName, /* attribute name                 */
      int         flags);   /* currently TCL_TRACE_WRITES only  */
```

Figure 7. Prototype for the OAT function that checks and triggers traces

For Tk widget and canvas item traces, which share the code to update attributes, the call to `Oat_CallObjTraces()` is inserted at the end of `DoConfig()`. This is the only place where the core code needs to be modified to support widget attribute traces in Tk. Figure 7 shows the prototype for function `Oat_CallObjTraces()` that specifies what information uniquely identifies the object and the attribute.

When the trace command is used in a Tcl script, the OAT code searches for the supplied subcommand keyword in the table of registered traceable types. When a matching keyword is found, a standard or custom function to manipulate a trace is executed. If the keyword does not match any of the registered traceable types, the trace command returns an error. This implementation supports traces on variables in a uniform interface, since variables are defined as a traceable type with subcommands "`variable`," "`vinfo`," and "`vdelete`," and with the Tcl-supplied C callback, `Tcl_TraceCmd()`, that processes these subcommands.

4. MediOAT: a traceable type extension for Continuous Media Toolkit

CMT [Rowe92], a multimedia toolkit from Berkeley, has an object system similar to Tk widgets. Examples of CMT objects are clocks, packet sources and destinations, and media play objects. Unlike Tk, however, the CMT object implementation does not use a single `ConfigureObject()` function; instead, each type of CMT objects parses its command options. For the OAT implementation of extended traces, this means that checks for traces need to be inserted in the attribute manipulation function for each type of CMT object. Based on our experiences with CMT multimedia applications, we decided that traces on clock attributes - speed and value - would be most useful. Traces on clock attributes trigger code when the time in a multimedia presentation "jumps", stops, or starts to move at a different rate. Our current version of MediOAT supports traces on attributes of CMT clocks. We wrote VCR control and jog shuttle control megawidgets based on MediOAT. Figure 8 shows GIF animation with VCR and jog shuttle controls. The control buttons - stop, play, fast forward, reverse - set the "`-speed`" attribute of the shared CMT clock. The trace callback, associated with clock speed updates, highlights the VCR buttons appropriately. The resulting code is compact; without traces, callback for each control button would have to know what other buttons to highlight.

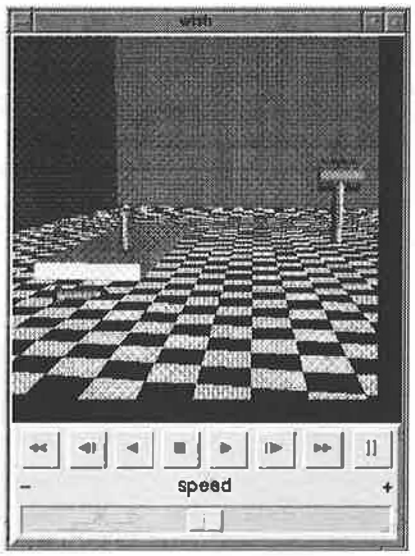


Figure 8. A MediOAT-based VCR controls (shown stopped: stop button disabled)

```
trace lts $lts -speed w \
    [list VCR_LTSChanged $id]
```

Figure 9. Creating a trace on a CMT clock; procedure `VCR_LTSChanged` manipulates the VCR buttons state.

5. OAT Implementation

Variable traces rely on special fields in the C structure describing a Tcl variable. This approach was not possible in OAT, since object systems currently do not reserve space for storing trace information. In Tk, there is not even a single data structure common to all widgets. We decided to use a hashtable associated with the interpreter to hold trace information. The hashtable string key consists of two components: the memory pointer of the C data structure describing the traced object, and the name of the attribute. To generate a string key, the binary pointer value is converted into its hexadecimal string representation. Why did we use the object pointer instead of name? Object name is not always available when its attributes are updated. To use the pointer for a part of the hashtable key, we assume that each traceable object is represented as a data structure with a unique address. We believe this assumption will hold as Tcl is upgraded with new versions. When the dual-ported object system was introduced in Tcl 8.0, we updated the OAT code to use the pointer to the object, rather than the string, representation in Tcl version 8 and above.

We did not observe performance degradation on trace manipulations from the OAT protocol. To evaluate the performance impact of OAT, we timed creation and querying of variable traces in the following two environments:

- original: Tcl7.6p2 with Jan Nijtmans's plus-patch [Nijtmans97], and Tk4.2p2 dynamically loaded
- OAT- and TkOAT-enhanced: Tcl7.6p2 with Jan Nijtmans's plus-patch, OAT, Tk4.2p2, and TkOAT loaded.

Timings were generated on several UNIX platforms: Solaris, Linux, and Irix. Both the original Tcl interpreter, and the interpreter with OAT and TkOAT loaded, produced essentially the same timings. We conclude that the lookup of additional trace keywords in the OAT library code does not lead to any noticeable performance degradation.

As we used OAT and TkOAT, we found it useful to trace components of object state that are not configurable attributes. The Listbox Pager example in Figure 1 relies on traces on such "virtual" listbox attributes: the number of listbox items, the number of visible items, and the first visible item. We concluded that supporting traces on these attributes was valuable enough to make further modifications to Tk: insert checks for traces in the listbox code. However, this violated our goal of non-invasive Tk extension. We hope that as traces and constraints are used more widely in Tcl/Tk applications, more attributes of widgets and objects will be "exported" by developers. The Tcl programmer will be able to use traces on a wider class of objects and attributes.

OAT was inspired by the variable traces in Tcl. TkOAT was made feasible largely by a single location in the Tk code where configurable widget and canvas item attributes are set. While MediOAT demonstrates that it is possible to insert OAT hooks in any place where object attributes are potentially updated, this defeats our goal of minimally invasive code changes. We encourage all extension developers who write object-like systems to adopt the Tk model, where names and types of object attributes are defined in a C data structure, and all attribute queries and updates go through function similar to Tk's `ConfigureWidget()`.

6. OAT and TclProp

While traces present a powerful abstraction for UI programming, they can be inconveniently low-level to be expressive. TclProp, a trace-based, script-only formula manager and data propagation engine for Tcl is described in [Iyengar95]. One of our goals in developing OAT was extending the benefits of TclProp formulas to arbitrary Tcl objects. TclProp was rewritten to be accommodate formulas on new types of objects. One main change to the TclProp API was the addition of traceable type names in formulas - these are needed to invoke the appropriate trace subcommands inside TclProp. TclProp uses the "`oat typeName`" command to retrieve the trace keywords, as described above. To support extended traces, TclProp also needs to know how to read and write attributes of traceable types. We use a Tcl array to associate traceable types with scripts that access attributes of these types. For example, the read code for the traceable type, Tk widget, is shown on Figure 10.

```
#the slot in the TP_traceableType array associates the traceable type
#name, widget, with the name of the procedure taking the names of a Tk
#widget and its attribute, and returning the code to read the value of
#this attribute.

set TP_traceableType(widget,read)  "widgReadFunc"

proc widgReadFunc {widgName attrName} {
      return "$widgName cget $attrName"
}
```

Figure 10. Extending TclProp to new traceable types: a Tk widget-specific attribute read procedure

```
TP_formula \
   "citem .canv rect1 coords" \                 # formula destination
      [list \                                   # list of formula sources
         "rC [list citem .canv rect coords]"] \       # source 1
   {list [expr [lindex $rC 0]+100] [lindex $rC 1] \   # formula code
         [expr [lindex $rC 2]+100] [lindex $rC 3]}
```

Figure 11. TclProp formula with Tk canvas item attributes as formula destination and source. Comments are for explanation only.

The Tcl interpreter evaluates procedure "`widgReadFunc`" when a TclProp formula is created, placing the code to read the value of the widget attribute into the formula code. Since the procedure is evaluated at formula creation time and not at formula propagation time, no additional performance penalty is incurred by the extension of TclProp to new traceable types.

Code for a TclProp formula typically accesses the values of variables and object attributes that the formula depends on. While variables reads in Tcl are compact, code to read the values of object attributes can be quite verbose. For example, to compute the width of a canvas rectangle, the following expression is needed:

```
{expr [lindex [.canv rect coords] 2] - \
      [lindex [.canv rect coords] 0]}
```

We enhanced TclProp formula syntax with *tags* to support a more compact access to object attributes in the formula code. Tags are associated with object attributes on which the formula depends, and are replaced with the actual attribute read code in the formula body. In the example above, if the tag "`rC`" is specified for the "`.canv rect coords`" rectangle attribute in the formula, the code above is simplified as follows:

```
{expr [lindex $rC 2] - [lindex $rC 0]}
```

The complete code for the TclProp formula ensuring that canvas rectangle `rect1` is offset by 100 pixels in x relative to `rect`, is shown on Figure 11.

In our experience with OAT and TclProp, we found the separation of trace detection and formula propagation very useful. A propagation model different from eager propagation model in TclProp can be built on top of Tcl and OAT traces. We experimented with several models, and implemented the lazy propagation alternative to TclProp, TclLazy, on top of OAT in a few hours. In the lazy propagation, variables in formulas are marked invalid on writes. A formula is evaluated only when an up-to-date value of its left-hand side is needed. The lazy model is appropriate when writes significantly dominate reads. We believe that the separation of traces and constraint propagation can be adopted in other scripting languages, such as Perl, to provide easier development of constraint engines.

7. Future Work

We use constraint programming and traces in our Tcl/Tk development extensively [Safonov96], and would like to see OAT and OAT-based extensions more widely used. We plan to work in the following four areas:

1. **Make more types of extension objects traceable**. We are currently considering adding traces to CMT media segment objects; this will facili-

tate the development of CMT-based timeline editors. We also plan to make objects traceable in VTk, a Tk-based 3D graphics extension, and GroupKit [Roseman96], a groupware for Tcl/Tk.

2. **Make adding traceability to objects easier**. Currently, the extension developer needs to modify code to insert checks for traces. While these changes are minimal, recompilation is still necessary. We plan to develop a protocol that will allow extension developers to insert hooks into the code that are later bound to the OAT function `Oat_CallObjTraces()`.
3. **Add object-oriented features to OAT**, based on [incr Tcl]. We see benefits in initializing class and object attributes to formulas rather than values, and in inheriting traces and constraints.
4. **Investigate other languages and environments for adding the trace protocol.** We have considered Perl as the potential target for adding the trace mechanism and trace-based constraint manager.

References

[Ellson96] John Ellson and Stephen North. TclDG - a Tcl Extension for Dynamic Graphs. In *Proceedings of the 4th Tcl/Tk Workshop*, p. 37. USENIX Assoc; Berkeley, CA, 1996.

[Iyengar95] Sunanda Iyengar and Joseph A. Konstan. TclProp: a data-propagation formula manager for Tcl and Tk. In *Proceedings of the 3rd Tcl/Tk Workshop*, p. 288. USENIX Assoc; Berkeley, CA, 1995.

[Nijtmans97] Jan Nijtmans. The Plus-patches. In *http://www.cogsci.kun.nl/tkpvm/pluspatch.html*

[Roseman95] Mark Roseman. When is an object not an object? In *Proceedings of the 3rd Tcl/Tk Workshop*, p. 197. USENIX Assoc; Berkeley, CA, 1995.

[Roseman96] Mark Roseman and Saul Greenberg. Building Real Time Groupware with GroupKit, a Groupware Toolkit. *ACM TOCHI*, March 1996.

[Rowe92] Lawrence A. Rowe and Brian C. Smith. A Continuous Media Player. In *Network and Operating System Support for Digital Audio and Video, Proceedings of the Third International Workshop on*, p. 376-86, 1992.

[Safonov96] Alex Safonov, Douglas Perrin, John Carlis, Joseph Konstan, John Riedl, and Robert Elde. Lessons from the Neighborhood Viewer: A Tool for Collaborative 3D Exploration of 2D Images. In *Proceedings of the 4th Tcl/Tk Workshop*, p. 203. USENIX Assoc; Berkeley, CA, 1996.

[Sah95] Adam Sah. Multiple Trace Composition and Its Uses. In *Proceedings of the 3rd Tcl/Tk Workshop*, page 288. USENIX Assoc; Berkeley, CA, 1995.

A Tk OpenGL widget

Claudio Esperança
COPPE, Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro
Cidade Universitária, CT, Sala H-319
Rio de Janeiro, RJ 21949-900, Brazil

Abstract

We present TkOGL, a Tk widget that enables the creation and display of 3D graphics using the OpenGL API. Our approach features a reasonably complete Tcl binding to the core OpenGL functionality, as well as a set of extensions that implement a higher-level interface to many common utility functions such as those provided by the OpenGL utility library (GLU).

1. Introduction

OpenGL [1] is becoming a standard Application Program Interface (API) for writing portable 3D computer graphics programs. On the other hand, the *Tk* toolkit offers a portable and powerful environment for the development of graphical user interfaces. It is to be expected then, that the merging of both capabilities should appeal to a wide audience. In fact, many attempts to do exactly that have been reported. Among these, we cite the *Tiger* system [2] and the *Togl widget* [3]. For various reasons, however, these packages did not meet my expectations. For instance, *Tiger* mimics the OpenGL API almost exactly, which makes the creation of simple 3D graphics unnecessarily complicated due to the inexistence of higher-level constructs. On the other hand, *Togl,* while providing the means to open a window for displaying OpenGL graphics, does not include Tcl bindings for any of the OpenGL rendering functions, thus forcing the user to program in C or C++.

Our primary purpose in writing the Tkogl widget was to enable both the experienced and novice users to generate and display 3D models in a concise manner. Moreover, the widget takes care of low-level tasks related to the embedding of OpenGL on a given window system, such as adjusting viewports to reflect window resize events and buffer swapping for double-buffered visuals.

The package was initially developed on an IBM RS-6000 workstation running AIX v3.2.5 and tested both with "real" OpenGL and with a free implementation of the OpenGL API, namely, the Mesa 3-D graphics library [4]. It was later ported to PCs running Microsoft's OpenGL implementation under Windows95. Currently, the package is known to work with Tcl 7.5/Tk 4.1 and Tcl 7.6/Tk 4.2. The distribution contains source code and Makefiles for some popular architecture/operating system combinations. In order to facilitate its installation in PCs, a pre-compiled DLL (dynamically loadable library) is also provided. The TkOGL home page address can be found at [5].

2. Design Issues

OpenGL [1] is a software interface to 3D graphics which was designed to provide optimum performance on client/server architectures. Thus, a typical application (client) consists of calls to OpenGL functions which are translated into messages that are sent to the graphics hardware (server), where they are interpreted and executed. Since it was designed to be portable to different architectures, the exact protocol involved in OpenGL messaging may vary. While this flexibilty is one of the strong points of OpenGL, in practice a usable OpenGL implementation must define several interface details in a non-portable manner. In particular, creating a window for displaying OpenGL graphics is non-trivial and eminently architecture-dependent. This task is further complicated when we consider that our aim is to embed such windows in frames managed by *Tk.* Some of the issues involved in this task are:

- How to create a child window which can be recognized by *Tk.*
- How to generate the architecture-dependent OpenGL runtime data structures (also called *contexts*).
- How to cope with window system-specific events. For instance, Microsoft's implementation of OpenGL only allows contexts to be created in response to an event which is not handled by the system-independent event dispatching mechanism of *Tk.*

- How to allocate other window system-specific data structures such as color maps.

These issues, although important, are not addressed further in this paper since they are not of interest to the typical user. The curious reader will be able to find some solutions to these problems by perusing the source code included in the Tkogl distribution [5].

Another important aspect concerning the integration of OpenGL and *Tcl/Tk* is the design of an appropriate set of Tcl bindings. Ideally, we would like an OpenGL widget to provide an interface which is similar to other *Tk* widgets. For instance, the `canvas` widget might serve as a model, since it provides the funcionality for generating 2D drawings. Unfortunately, however, 3D graphics are substantially more involved, and the overall approach used by `canvas` (i.e., to provide a few graphical item types such as rectangles and ovals which can be created and configured) cannot be employed in quite the same manner.
The rest of this paper describes our approach to this problem.

3. A Simple Application Using Tkogl

The integration between OpenGL and Tk is achieved by a `package` called `Tkogl`, which in Unix-based installations is statically linked in the extended Tcl/Tk windowing shell called `glwish`. Under Windows95, the package can be dynamically loaded by executing a corresponding `package require` command. In any case, a Tcl script which uses the package should include the following line:

```
package require Tkogl
```

Once it is ascertained that the package is loaded, one or more windows can be created for displaying OpenGL graphics. Such windows can be created in a similar way to other Tk widgets by using the `OGLwin` command, which has the following format:

`OGLwin` *pathName ? option ... ?*

where each *option* can be one of the following:

`-accumsize` *accumSize* specifies that the accumulation buffer should support *accumSize* bit planes for each of the red, green and blue components. If an alpha component for the color buffer has been requested, the same number of bit planes is also requested for the alpha component of the accumulation buffer. By default, no accumulation buffer is requested.

`-alphasize` *alphaSize* specifies that the color buffer should support *alphaSize* bit planes for the alpha component. By default, no alpha bit planes are requested.

`-aspectratio` *ratio* forces the viewport of the window to the width/height fraction given by *ratio*, which should be a positive floating point number. The viewport is then defined as the biggest possible rectangle with the specified aspect ratio centered inside the window. If *ratio* is 0.0 (the default), no aspect ratio is enforced, which means that the viewport will always take the same shape as the window.

`-context` *pathName2* makes the OpenGL context of *pathName* share display lists with that of *pathName2*, which should also be the name of an OGLwin widget.

`-depthsize` *depthSize* specifies the number of bit planes for the depth buffer (also called *z-buffer*). By default, this number is 16. A *depthSize* of 0 means that no depth buffer is required.

`-doublebuffer` *doubleFlag* specifies whether or not a double buffered visual will be used (true, by default).

`-height` *height* specifies the height of the window in pixels. Default:300.

`-stencilsize` *stencilSize* specifies the number of bit planes requested for the stencil buffer (zero, by default).

`-width` *width* specifies the width of the window in pixels. Default:300.

Currently, OGLwin can only be used to create windows which will use the RGBA color model. By default, OGLwin creates a double-buffered RGB window with the biggest number of bitplanes supported by the current software/hardware environment. The configuration options described above can be used to allocate additional buffers, e.g., an accumulation or a stencil buffer. If the requested buffers cannot be allocated, then OGLwin fails, producing a standard Tcl error result.

An OpenGL window is typically created for visualizing a series of graphical objects. In most window systems, the contents of the window must be redrawn every once in a while, for instance, when the window is resized or deiconified. Usually, quite a few OpenGL rendering commands must be executed in order to reproduce the contents of the window. Although we aim to be able to generate any OpenGL command from within a Tcl script, it would be very time-consuming to interpret a very long sequence of Tcl commands every time a given OpenGL window needed to be redrawn. Fortunately, OpenGL offers a display list capability, whereby several commands can be pre-compiled and stored in the display server, ready to be re-executed as needed. Thus, a sensible management of an OpenGL window (such as the one created by the `OGLwin` command) is to reserve a display list which will contain all rendering commands that are to be executed whenever the window needs to be redrawn. In this document, we refer to such a list as the *main list*. In addition to calling the main list whenever a redraw is needed, the widget issues **glFlush** command and takes care of swapping the front and back buffers (when a double-buffered visual is being used). The contents of the main display list can be redefined by means of the `main` widget command, which has the following format:

pathName `main` *? option ... option ?*

where

pathName	is the name of an OpenGL window.
option	is one of the OpenGL commands currently supported by the package. These will be described later on.

The program listing in Example 1 below shows a very minimal script that creates a window to display a triangle. The display produced by that program is shown in Figure 1.

```
package require Tkogl
OGLwin .gl
pack .gl
.gl main -clear colorbuffer \
   -begin triangles \
   -vertex -1 -1 \
   -vertex 0 1 \
   -vertex 1 -1 \
   -end
```

Example 1: A simple script to display a triangle.

Notice that the script above relies on several variables of the OpenGL state machine having their initial default values. For instance, the default value of the **Color** state is white, while the the default value of the **ClearColor** state is black, which means that the triangle will be drawn in white over a black background.

Instead of using the main display list mechanism for keeping the window updated, it is also possible set up a script to be executed every time an *Expose* event is caught by Tk. In this case, instead of using the main widget command to set up the main display list, the

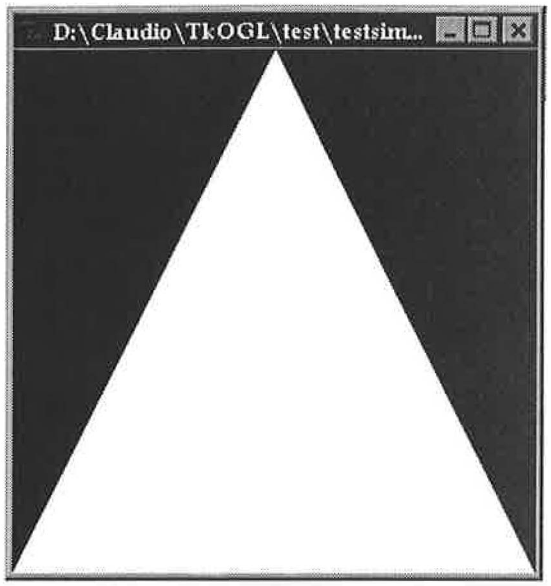


Figure 1: Display produced by the script of Example 1

same OpenGL commands can be issued by means of the `eval` widget command, which has the following syntax:

pathName `eval` *? option ... option ?*

where *pathName* and *option* have the same meanings as in the `main` command.

For instance, it is possible to rewrite our minimal script to catch Expose events directly. This is shown in Example 2 below.

It should be noticed that the default display list mechanism is usually superior to catching events and redisplaying the picture. This is because in the former case all OpenGL commands are already stored in a display list in the server, while in the latter case, all commands must be reinterpreted and transmitted from the client to the server every time the window must be redrawn.

```
package require Tkogl
pack .gl
bind .gl <Expose> {
   .gl eval -clear colorbuffer \
      -begin triangles \
      -vertex -1 -1 \
      -vertex 0 1 \
      -vertex 1 -1 \
      -end
}
```

Example 2: Displays a diagonal line by catching *Expose* events and redrawing the picture with the `eval` widget command.

4. OpenGL option commands

Many OGLwin widget commands (e.g., `eval`, `main`) require a list of options that denote OpenGL commands. The overall format of such options is

glCommandName ? arg ... arg ?

where

glCommandName is a Tcl string that denotes an equivalent OpenGLcommand. The string corresponding to a given OpenGL procedure is the name of that procedure stripped of its **gl** prefix and of eventual data type suffix. Mixed upper- and lowercase characters can be used. Thus, for instance, procedure **glMatrixMode** corresponds to option `-matrixmode` (other lower/uppercase combinations such as `-MatrixMode` are also acceptable), procedure **glColor3f** corresponds to option `-color`, and so on.

arg is a Tcl string equivalent to an argument in the corresponding OpenGL command. The following rules are useful to determine how OpenGL procedure arguments are mapped into equivalent Tcl strings:

- Arguments of type **GLenum** are mapped into a string with the same spelling as that of the equivalent constant, except that the GL prefix is dropped, as well as any underscore ('_') characters. Mixed upper- and lowercase characters can be used. For example, constant **GL_DEPTH_TEST** may be written either as `depthtest` or `DepthTest`.
- Numeric arguments are represented by equivalent Tcl strings. Integer types (e.g. **GLint**, **GLuint**) are parsed as integer Tcl values and floating-point types (e.g., **GLfloat**, **GLdouble**) are parsed as floating-point values.
- When the same OpenGL function supports both integer and floating-point variants of the same function, the floating-point (**GLfloat**) variant is implemented. For example, command

  ```
  -color 1 0 0
  ```

  is the same as

  glColor3f (1.0, 0.0, 0.0);

- If an OpenGL procedure requires a vector argument, this is supported by spelling out the contents of the vector as discrete *arg*'s. For instance, the "C" code fragment

  GLfloat ctrlpoints [4][3] = {
  {-4.0, -4.0, 0.0}, {-2.0, 4.0, 0.0},
  {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}
  }

  glMap1f (**GL_MAP1_VERTEX_3**,
  0.0, 1.0, 3, 4, &ctrlpoints[0][0]);

  would be translated into Tcl as

  ```
  -map1 map1vertex3 0 1 3 4 \
     -4 -4 0  -2 4 0 \
     2 -4 0   4 4 0
  ```

- In the case of procedures such as **glClear**, which require bit masks as arguments, the individual bit mask constants are mapped to strings in much the same way as **GLenum** constants, except that the **_BIT** suffix is also dropped. Furthermore, the bit mask is assumed to be a bitwise "or" ofall *arg*'s. For instance,

  **glClear (GL_COLOR_BUFFER_BIT |
  GL_DEPTH_BUFFER_BIT);**

  becomes

  ```
  -clear colorbuffer \
         depthbuffer
  ```

Most OpenGL procedures have equivalent option commands. In a few cases, the argument lists of an option command and its associated OpenGL procedure have slightly different argument lists, chiefly those that deal with textures and images. Also, some procedures of the OpenGL Utility Library (**glu**) were also implemented as option commands. The complete list of option commands can be found in the documentation included in the Tkogl distribution [5].

5. OGLwin widget commands

Although most of OpenGL's capabilities could be exercised by using the `eval` and `main` widget commands, we felt that certain common tasks might be more easily programmed if a suitable set of additional commands were provided. For example, the OpenGL Utility Library (**glu**) provides several functions that can be used to render quadric surfaces such as spheres, cones and cylinders. These functions control many aspects of the rendering process, such as the dimensions of the surface, whether texture coordinates should be generated, etc. This functionality can be captured in a more natural way within Tcl scripts with a corresponding widget command corresponding to the surface type to be rendered. Consider for instance the `sphere` widget command described below:

pathName `sphere` *?*`-displaylist` *dlist?* *?*`-normals` *normals?* *?*`-drawstyle` *drawStyle?* *?*`-orientation` *orientation?* *?*`-texture` *texture? radius slices stacks*

Renders a sphere using the GLU facilities for quadrics (refer to the **gluCylinder** function). By default, the rendering is compiled into a new display list whose number is returned as the result of the widget command. If a display list number *dlist* is specified by means of the `-displaylist` option, then that list is used. As a special case, if *dlist* is specified as `none`, the rendering is performed immediately, and no display list is generated or overwritten. The remaining options correspond to rendering styles as implemented by functions **gluQuadricNormals**, **gluQuadricDrawStyle, gluQuadricOrientation** and **gluQuadricTexture**, respectively. The possible option values are strings derived from corresponding symbolic constants. Thus, for instance, `-normals flat` corresponds to calling **gluQuadricNormals** with an argument equal to **GLU_FLAT**.

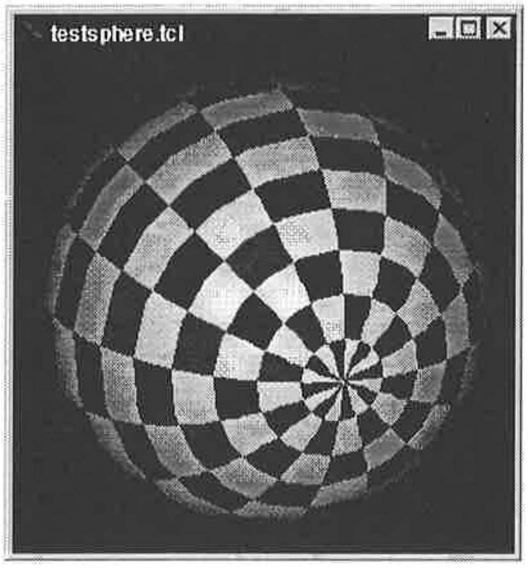


Figure 2: Display produced by the script of Example 3

The `sphere` command encapsulates all the functionality of the **glu** library procedures for rendering spheres. This is illustrated in the script labeled "Example 3" below which renders a shaded, textured sphere (see Figure 2).

```
package require Tkogl
# Create a checkerboard image
image create photo tmp -width 2 \
   -height 2
image create photo img -width 64 \
   -height 64
tmp put {{white red} {black white}
img copy tmp -zoom 4 4 -to 0 0 64 64

# Create an OpenGL window
pack [OGLwin .gl]

# Configure point of view and texture
.gl eval \
    -matrixmode projection \
    -ortho -2 2 -2 2 -2 2 \
    -matrixmode modelview \
    -rotate 30 1 1 0 \
    -enable lighting \
    -enable light0 \
    -enable depthtest \
    -enable texture2d \
    -texparameter texture2d \
     texturewraps repeat \
    -texparameter texture2d \
     texturewrapt repeat \
    -texparameter texture2d \
     texturemagfilter nearest \
    -texparameter texture2d \
     textureminfilter nearest \
    -texenv textureenv \
     textureenvmode modulate\
    -teximage2d 0 0 img

# Create sphere object
set quadric [.gl sphere -texture yes \
   -drawstyle fill -normals smooth \
    1.8 20 20]

.gl main -clear colorbuffer depthbuffer\
    -call $quadric
```

Example 3: Displays a textured shaded sphere.

Note in Example 3 that the texture image was created by means of *Tk*'s `photo` extensions. The Tkogl package interacts with images created with Tk in order to provide a smooth integration with OpenGL capabilities. Thus, the argument list of OpenGL function **glTexImage2d** was slightly modified in the corresponding Tkogl option command `-teximage2d` so that an image name could be used instead of an array of bytes.

The Tkogl package implements several widget commands that encapsulate capabilities usually accessed by means of the **glu** library such as the rendering of quadric and NURBS surfaces, polygon tesselation, etc. Additional widget commands are also included to implement other useful rendering and modeling extensions not supported by the **glu** library. For example, Tkogl includes the `gencyl` command, which supports the creation of generalized cylinders (i.e., objects obtained by sweeping a two-dimensional shape along a curve in 3D). Figure 3 depicts one of these objects obtained with a demo application included in the distribution.

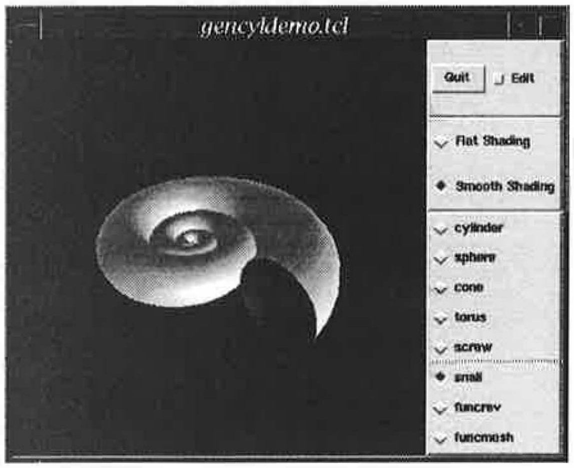


Figure 3: Generalized cylinder object obtained with a demo application.

6. Input Events

One of the nicest features of Tk is the simple way user-input events can be handled by means of the `bind` command. The OpenGL standard, on the other hand, offers limited facilities for managing input. These facilities, however, were included in the Tkogl package and can be easily put into use within a Tcl script. For instance, one of the most bothersome difficulty lies in establishing the correspondence between window coordinates and world coordinates. The GLU library provides two functions – **gluProject** and **gluUnProject** – for exactly that purpose. While these functions require a long list of arguments which include the current viewport and transformation matrices, their Tkogl counterparts – the `project` and `unproject` widget commands – only require three arguments representing the three coordinate values of the point to be transformed. Example 4 below demonstrates the use of the `unproject` command in a program for drawing lines. Notice how points which are input by clicking the mouse button are captured with a `bind` command and passed into procedure `newvertex` which computes the corresponding world coordinates. A sample output of this program is shown in Figure 4.

```
package require Tkogl

pack [OGLwin .gl]

proc newvertex { x y } {
    global vertices
    append vertices -vertex \
      " [.gl unproject $x $y 0] "
    eval .gl main -clear colorbuffer\
      -begin lines $vertices -end
}

.gl main -clear colorbuffer

bind .gl <Button-1> {newvertex %x %y}
```

Example 4: A simple line drawing program.

Other OpenGL facilities for handling user input are similarly supported in Tkogl. In particular, object selection and "picking" can be handled in Tkogl through the use of the `select` command. This command takes care of allocating a hit buffer and processing the list of hit objects, returning a single Tcl list which can then be easily parsed within the script.

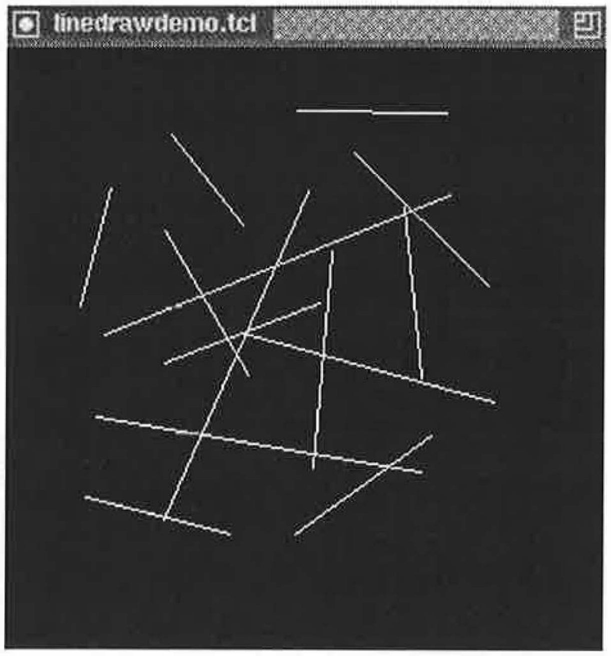


Figure 4: Sample output of the script "Example 4".

7. Conclusions

We described our implementation of a Tk widget for generating and displaying 3D graphics through the use of the OpenGL API. Our approach, in contrast with similar ports of OpenGL to the Tcl/Tk environment, combines the accessibility of most OpenGL functions through widget commands and options with a repertoire of extensions that enable users to model several objects with compact Tcl scripts.

References

1. J. Neider, T. Davis, M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*, Addison-Wesley, Reading, Massachusetts, 1993.
2. Tiger 1.2 ftp site. URL: `ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/TIGER1.2`
3. Togl home page. URL: `http://www.ssec.wisc.edu/~brianp/Togl.`
4. Mesa home page. URL: `http://www.ssec.wisc.edu/~brianp/Mesa.html`
5. TkOGL home page. URL: `http://aquarius.lcg.ufrj.br/~esperanc/tkogl.html`
6. OpenGL Architecture Review Board, *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1*, Addison-Wesley, Reading, Massachusetts, 1992.

The ImageTcl Multimedia Algorithm Development System

Charles B. Owen
Dartmouth Experimental Visualization Laboratory
Dartmouth College
6211 Sudikoff Laboratory, Hanover, NH 03755
cowen@cs.dartmouth.edu
http://devlab.dartmouth.edu/imagetcl/

Abstract

The IMAGETCL *multimedia development system is a new Tcl/Tk-based development environment specifically targeting development of high-performance multimedia data analysis algorithms. Multimedia algorithm development is complicated by large volumes of data, complex file formats, compression and decompression, and temporal synchronization. Testing algorithms requires elaborate user interfaces which can display intermediate and result images, play audio, and adjust parameters.* IMAGETCL *uses the features of the Tcl/Tk environment as a base on which to build a system that significantly aids this process. This paper describes the* IMAGETCL *approach to algorithm development and describes several applications of* IMAGETCL *in the Dartmouth Experimental Visualization Laboratory (DEVLAB).*

1 Introduction

IMAGETCL is a new system designed to support multimedia algorithm development. Multimedia algorithm development can be described as a five step process: (1) algorithms are theoretically devised, (2) a prototype of the algorithm is implemented, (3) test procedures are devised to test and debug the algorithm, (4) algorithm performance and effectiveness are tested using standard and custom data sets, and (5) user interfaces are developed to support user interaction and performance demonstration. These steps are a process and are not necessarily sequential. Media data (images, video, text, etc.) are noisy, imprecise, and ambiguous. The one perfect procedure for motion analysis, media alignment, or speech recognition is not known. Hence, proposed algorithms require extensive testing and adjustment.

The primary design goal of IMAGETCL is simplifying this process. IMAGETCL is based on the Tool Command Language and the Tk Toolkit and has been designed to provide the algorithm developer with a rapid prototyping environment which greatly aids steps 2 through 5 [7]. Automatic tools, a powerful function and application library, a cohesive media manipulation structure, and a compiled high-performance environment support the algorithm development process. Test procedures can be quickly and easily devised and modified using scripts written in Tcl. A central database for a site can be built which contains test data for algorithm evaluation, and high quality user interfaces can be quickly devised using Tk and Tix.

IMAGETCL is unique in its emphasis on **algorithm development in C++**. A large number of components are available at the script level and can be used to build and test applications. However, experience in the DEVLAB has shown that interpreted scripting languages do not lend themselves well to algorithms that directly manipulate samples, pixels, or voxels. Constraining the developer to matrix manipulations and avoiding low level loops as is common in systems such as Matlab limits flexibility in the design process [14]. IMAGETCL uses Tcl/Tk for what is does best, overall control and configuration, and leaves the low level processing in C++.

1.1 Related Work

Most multimedia development environments have focused on providing high-level scripting tools in an attempt to eliminate compiled language development. This approach assumes that such a complete set of tools can be devised and that the interpreted scripting code can use such powerful primitives that performance will not be compromised. A major element of the IMAGETCL design philosophy is that such a goal is difficult, if not impossible, to achieve. Systems often provide some facility for extending

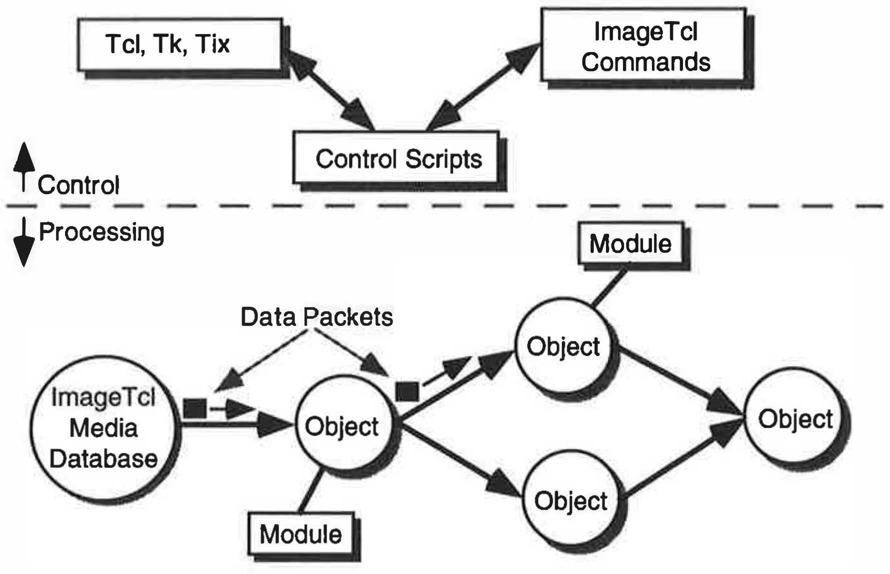


Figure 1: IMAGETCL Execution Structure

the environment using a compiled language (usually C). In IMAGETCL, however, easy extension of the environment using C++ is a basic development element.

Several systems have influenced IMAGETCL development. VideoScheme, also developed at the DEVLAB, utilized the Scheme programming environment as a rapid prototyping system for digital video editing and analysis [6]. MIT's ViewStation is a powerful multimedia environment which is also Tcl/Tk based [3]. ImageTcl uses a data-flow structure very similar to that of the ViewStation. Cornell's Rivl system abstracts away most of the media parameters, including resolution and timing, and provides very powerful media manipulation tools [13]. ImageTcl uses an interpreter level data representation similar to that of Rivl. Multimedia *application* development is the major goal of these systems. IMAGETCL is designed primarily for *algorithm development*, though it has many application level features.

2 System Overview

The system design of IMAGETCL can be divided into an execution structure and a development structure. Figure 1 illustrates the execution structure, and Figure 2 illustrates the development structure. This section describes the components in these figures. Note the clear delineation between *control* and *processing*. IMAGETCL performs media processing at the compiled code level and control and user interfaces at the Tcl scripting level.

2.1 Media Processing Model

IMAGETCL is based on a dataflow media processing model. This is illustrated in the processing section of Figure 1. A *directed multigraph* (henceforth referred to simply as a *graph*) models the flow of data in the system. The nodes of the graph are *objects*, which provide general media processing. Objects are connected with directional edges representing the flow of *data packets* containing media data. *Modules* attach to objects and provide a specific algorithm implementation. The use of modules allows the separation of generic algorithm code from specific algorithm code. As an example, *optical flow* is a generic class of algorithm which attempts to derive flow fields describing the motion in an image sequence. Horn and Shunck devised a particular algorithm for computation of optical flow [2]. In the IMAGETCL model, optical flow is an object and the Horn and Shunck algorithm is a module.

Composite objects are constructed using Tcl scripts. The IMAGETCL *Media Database* (ItMDB) in Figure 1 is an example composite object. A library of these scripts is available to the application developer. Components in that library appear as objects in the graph. Composite objects are used in IMAGETCL to hide issues of file formats and data compression and decompression from the developer. Media data does not flow through interpreted code

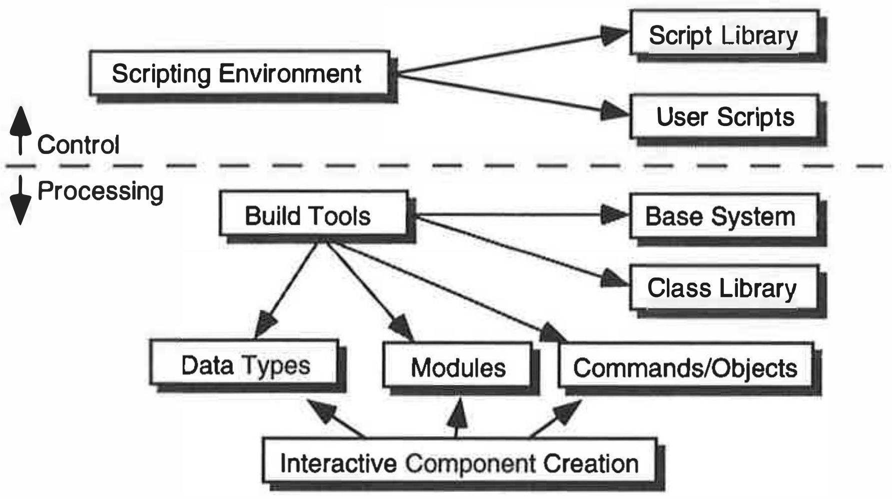


Figure 2: IMAGETCL Development Structure

by default in this model[1]. Construction of the graph and all control is implemented in Tcl. This approach allows for very fast development of test scripts that provide a node with data and collate and present results. The scripts do not process the actual data and control is not embedded in compiled code, balancing the flexibility of Tcl with the performance of compiled languages.

2.2 Core and Standard System Components

IMAGETCL components can be divided into three sets: (1) core components, (2) standard components, and (3) user components. Multimedia development systems are subject to constant expansion and can rapidly become large and cumbersome. Compile and link times can grow unwieldy. IMAGETCL is highly modularized with components treated as building blocks that can be chosen at will. Both applications and dynamic link libraries can be built using any desired set of components. Only one line in an IMAGETCL build file need be changed to add a component.

Core components are those necessary for system operation. These components are always included. *Standard components* are those which are included in the published system. A standard system containing all of these components is available. A user can also construct a subset system with only desired components. This modularity allows for faster compile and link times and has been particularly useful in system development.

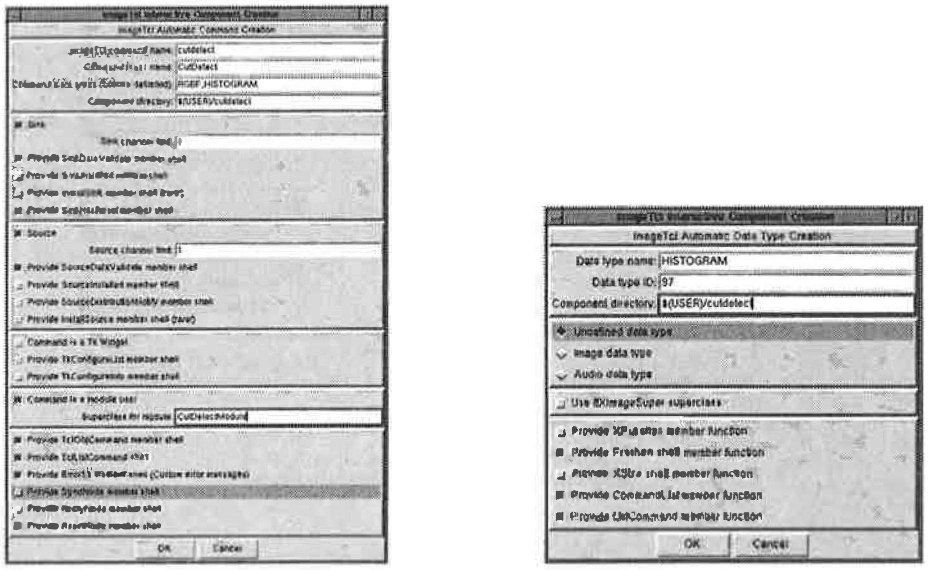
[1]An alternative interpreted path exists for application flexibility.

The `itbuild` utility builds make files for applications and libraries. This utility can combine not only system and user components, but also outside components. `itbuild` automates make file creation, system configuration, and dependency checking.

2.3 User Component Creation

The most important feature of IMAGETCL is user component creation. Every effort has been made to simplify this process. The *Interactive Component Creation Utility* is the first step in component creation. A user can easily add new commands, objects, modules, and data types to the system. Figure 3 illustrates the Interactive Component Creation Utility fill-in forms for new commands and data types. Characteristics of the component are entered into the form and all necessary files to create the new component are generated. IMAGETCL is C++ based and has superclasses for a variety of components. These classes have many options and creating new elements manually would be cumbersome. The algorithm designer is primarily interested in implementation of the algorithm, not class construction, so automatically generated files provide a template that can be used immediately to create a new component.

Three files are created: a header file (.hxx file), a source file (.cxx), and an IMAGETCL build file (.itb). The header and source files implement all classes necessary for the object including template procedures. Example code for the function is included in comments. The IMAGETCL build file is used with the system build tools to include the component. The new component can be immediately included in the


ImageTcl Interactive Component Creation

ImageTcl Automatic Command Creation

[illegible] name: cutdetect
[illegible] name: CutDetect
[illegible]: RGBF,HISTOGRAM
Component directory: $(USER)/cutdetect

Sink
Sink channel limit: 1
Provide SinkDataValidate member shell
Provide [illegible] member shell
Provide [illegible] member shell
Provide [illegible] member shell

Source
Source channel limit: 1
Provide SourceDataValidate member shell
Provide SourceInstalled member shell
Provide Source[illegible]Modify member shell
Provide InstallSource member shell (rare)

Command is a Tk Widget
Provide TkConfigureList member shell
Provide TkConfigureInfo member shell

Command is a module user
Superclass for module: CutDetectModule

Provide TclObjCommand member shell
Provide TclListCommand shell
Provide Error[illegible] member shell (Custom error messages)
Provide Synchronize member shell
Provide [illegible] member shell
Provide [illegible] member shell

OK Cancel

ImageTcl Interactive Component Creation

ImageTcl Automatic Data Type Creation

Data type name: HISTOGRAM
Data type ID: 97
Component directory: $(USER)/cutdetect

◆ Undefined data type
◇ Image data type
◇ Audio data type

Use [illegible]ImageSuper superclass

Provide [illegible] member function
Provide Freshen shell member function
Provide XSize shell member function
Provide CommandList member function
Provide ListCommand member function

OK Cancel

Figure 3: ImageTcl Interactive Component Creation Utility

system, compiled, and linked. The user must still, of course, provide algorithm functionality, but all of the mechanics of providing a new Tcl command, instantiation of objects, and usage will be fully operational.

2.4 Algorithm Support

Many resources are available at the algorithm implementation level. Class functions simplify use of the ImageTcl environment. These functions include command processing, error management, and command dispatching. Support for lists, stacks, directed graphs, trees, and other common containers is provided in easy to use template classes. Matrix and vector classes are provided and include overloaded operators for arithmetic manipulation as well as many standard signal processing and statistical analysis functions. A library of Tcl scripts is provided to simplify test and application level development.

3 ImageTcl Algorithm Development Process

Section 1 listed the five steps in the multimedia algorithm development process: (1) algorithm design, (2) prototype implementation, (3) implementation testing, (4) performance and effectiveness testing, and (5) user interface development. This section illustrates the ImageTcl approach to each step.

3.1 Algorithms are theoretically devised

Algorithm design is not directly supported in ImageTcl. The user is expected to design the algorithm. However, a common algorithm design aid is the analysis of data for statistical and visual characteristics. ImageTcl tools can be applied to this process.

3.2 Prototype implementation

An algorithm is implemented as new ImageTcl components, typically objects and modules. The new components must be derived from one of several C++ superclasses, must register themselves in the system, and must implement various virtual functions. While all of this information is documented, it would slow implementation considerably if this detail was the responsibility of the programmer. Instead, the programmer uses the ImageTcl Interactive Component Creation utility described in section 2.3. Necessary features, class and command naming, and function selection for the new component are described in simple fill-in forms. A complete set of template files for the new component is generated automatically.
The new component must then be added to the system. The user creates an ImageTcl build file for a system–this is a simple file which lists the components to be used and some build options such as optimization and build type. Adding a new component to the system requires a single line in the build

file. The **itbuild** utility reads the build file and creates a makefile and an application initialization file. All that is necessary are **make depend** and **make** to use the new component. Users never modify make files directly in IMAGETCL.

The created components are ready to compile and link, though they are non-functional. The user "fills-in" the algorithm. Example code included in comments simplifies this process. The default behavior is to create an application which will utilize the standard IMAGETCL shared library.

3.3 Test procedures

Once the new components are created, test scripts are required which will aid in the debugging of the components and ensure they are functioning correctly. A common structure for components is enforced by the system, simplifying script development. A *command* creates a named object. As an example, the statement **imagerotate rot** will create a new object named **rot**. The command (**imagerotate**) is equivalent to a C++ class and the object an instantiation of the class (in fact, this is the underlying implementation). More details on the command interface are included in section 4.2. Various test procedures can be devised for the same algorithm (and executable), and test procedures are high-level and can be developed quickly.

A common element in multimedia algorithm testing is parameterization. Many algorithms require thresholds, iteration counts, and other execution parameters, and often the adjustment of these parameters is a major part of the research. Parameters in IMAGETCL development are defined at the scripting level. Support for parameter input from the scripting level can be implemented using as few as two lines of C++ code in the algorithm implementation. Because parameterization is implemented in interpreted Tcl scripts, parameters can be adjusted quickly, either by editing the script and re-executing, or by adding Tk user interface components. This rapid development of test procedures is a significant advantage of using the Tcl environment.

3.4 Performance and effectiveness testing

Multimedia algorithms (and information retrieval algorithms in general) require validation on large sets of data. An algorithm can be tuned to work well on a small data set, but must demonstrate effectiveness and performance on larger data sets. The IMAGETCL *Media Database* allows accumulation of standard test data for a development site. This data is then available to all IMAGETCL users. The database contains pointers to media files which can be located on different file systems.

3.5 User interface development

Though user interface development is listed last, it is commonly concurrent with most other steps. A project's user interface typically moves from minimal functionality to extensive functionality as the project proceeds from small tests to large scale validation. The Tcl/Tk/Tix environment, in combination with composite objects in the IMAGETCL library, tremendously simplifies user interface development. User interface design is often highly iterative, with components adjusted and rearranged. The placement of this process at the scripting level has been highly advantageous.

4 System Structure

IMAGETCL is highly modular with pieces designed as building blocks which can be added and removed at will for simpler system debugging, porting, and expansion. These blocks build upon the *core system* and the IMAGETCL *environment*. A standard interface to components is enforced by the system in order to simplify component development and user scripting. A segregated set of *machine specific components* is included to support features specific to a single hardware architecture. A set of standard libraries, both Tcl and C++, as well as standard user algorithms are integral elements of the system.

4.1 Directory layout

An IMAGETCL system is located in a directory available to all users and has a structure as illustrated in Figure 4. Machine and operating system specific directories allows segregation of system binaries and platform specific components. An example machine specific component would be the interface to audio and video machine hardware. This structure provides all IMAGETCL source and binaries to all users.

4.2 Components interface

IMAGETCL enforces a common component interface. Commands are grouped into two categories: object commands and non-object commands. *Non-object commands* provide control features in the system

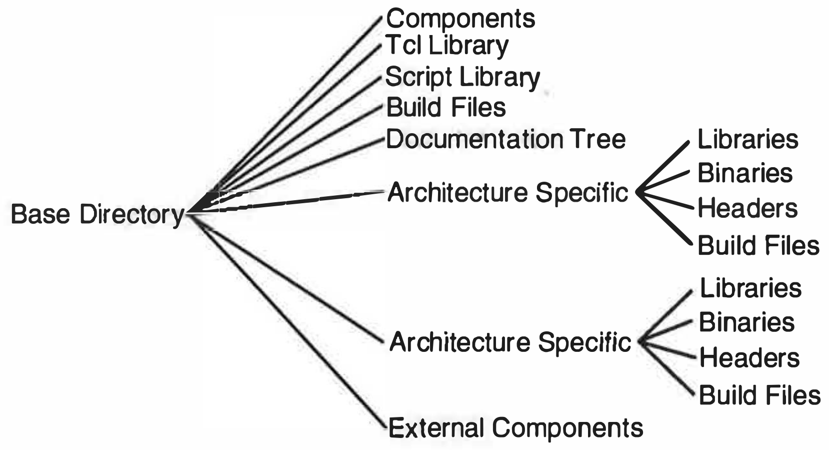


Figure 4: IMAGETCL Directory Structure

which do not entail the instantiation of an object. These are elements which have a single persistent object associated with them. An example is the **connect** command, which is used to connect object outputs to inputs in the graph. An example usage of this command is: **connect video 0 display 0**. In this example output channel 0 of a video input device is connected to input channel 0 of a display device. Note that *video* and *display* in this example are object names, not commands[2].

The system is highly object-oriented and the majority of commands are *object commands*. An object command creates an object. As an example, the **audiooutput** command can create an object named **audio** using the statement **audiooutput audio**. The object is then available as a Tcl command in the system. Communication with the object is through use of this command. This is identical to the widget structure in Tk.

The structure for parameters on commands is also similar to Tk. Parameters which begin with a "-" are considered to be *object list commands*, usually shorted to just *list commands*. As an example, in the Tcl statement **audio -sink 0 PCMW**, the **-sink** is a list command followed by parameters on that command. Multiple list commands can be used on the same line; IMAGETCL automatically scans this list and provides the command and options to the component. An example of a list command processing member function is included in Figure 5. This example processes the list command **-angle** with argument count checking.

List commands are used when no result is required. *Object commands* are used when a response is required. Only one object command is allowed per line and object commands do not start with "-". An example object command usage might be **set height [videocapture height]**. Figure 6 illustrates the member function code to support this command. **TclResult** is a member function of the **ImageTclEnvUser** superclass.

[2]As in Tk, they are now Tcl commands, though they are referred to in this system unambiguously as objects.

```
int ImageRotate::TclListCommand(char **argv,
                                     int argcnt)
{
   // Process the command
   if(strcmp(argv[0], "-angle") == 0)
   {
      if(argcnt != 1)
          return ER_ARGCNT;

      angle = atof(argv[1]);
   }
   else return ER_BADLISTCOMMAND;

   return ER_NONE;
}
```

Figure 5: TclListCommand Member Function

Commands are dispatched to modules attached to objects, then to the objects, then to generic handlers in superclasses. This structure allows generic object and standard system commands with minimal programmer intervention.

4.3 Data types

IMAGETCL uses a data passing structure very similar to the ViewStation's [3]. A *data packet* is passed along the edges of the directed graph. Each edge has an associated packet queue. A *data buffer* is as-

```
int VideoInput::TclObjCommand(int argcnt,
                              char **argv)
{
   if(strcmp(argv[1], "height") == 0)
   {
      sprintf(TclResult(), "%d", VideoHeight());
      return ER_NONE;
   }

   return ER_BADCOMMAND;
}
```

Figure 6: TclObjCommand Member Function

sociated with a data packet and contains the actual media data. As in the ViewStation, a data packet is a general control object. Temporal sequencing information is contained in the data packet. Data buffers are specific to a data type (derived, of course, from a common superclass) and serve as a payload. Modifying sequencing information (reordering frames, changing frame duration, etc.) can be done by creating a new data packet and associating the same data buffer with that packet, avoiding duplication of large media data. Some example data types in IMAGETCL include monochrome and color images, PCM audio, JPEG compressed images, and arbitrary vectors and matrices.
A unique feature of IMAGETCL is that data types are plug-in components. Users can create new data types at will. Some example user data types have included speech biphone probabilities and color image histograms.

4.4 System components

System components are standard plug-in elements of IMAGETCL. These include audio and video input/output, media conversion, standard media manipulations, arbitrary data packet routing, image display, and compression and decompression. A complete list is omitted here due to space considerations.

4.5 Applications and libraries

A large library is included in IMAGETCL. System class libraries include matrix manipulation, strings, lists, and a directed graph template class. At the scripting level numerous scripts have been designed which are available as library components. One example is the *ItViewer* library. Many applications are tested through the application of media in specific formats. ItViewer creates a simple media browser with typical temporal controls. However, this browser is designed as a system component. New controls and menus can be added and data packets can be routed to algorithms under test. Most initial test applications are built on the ItViewer library. Initial test scripts can often be only a few lines long.
In addition to library components, many script applications are available, including development and media manipulation tools. These include applications which view media in different formats such as three-dimensional volume slices, and medical image viewers which support contrast adjustments.

4.6 Why Tcl?

Numerous scripting environments were examined prior to choosing Tcl for this project. A previous DEVLAB research system, VideoScheme, utilized the Scheme programming language, specifically the *Scheme in on defun* (SIOD) implementation [6]. However, if was felt that the functional nature of Scheme was not a natural description of the procedural construction and control of test environments. Perl was considered as an environment as well [15]. The large number of specialized variables and the lack of a standard user interface were reasons for not choosing that environment. Other choices considered included creation of an ad-hoc scripting environment. Tcl/Tk has clearly been an excellent choice.

4.7 Why C++?

IMAGETCL development is C++ based. Since the Tcl/Tk environment is written in the C programming language, an obvious question is: why C++? C++ has been chosen as the underlying development language for several reasons [12]. The fundamental basis for this choice is that the object-oriented structure of C++ matches well with the media processing model used in the system. In addition, the strong typing features of the language are felt to be advantageous by the authors in that they decrease errors due to type mismatches and lack of function prototypes. The mix of Tcl and C++ has been seamless and quite successful in this system.

5 Applications

The DEVLAB has applied IMAGETCL to numerous application areas in multimedia data analysis [4, 10, 9]. This section briefly describes a few of these projects, illustrating the specific advantages of the Tcl/Tk approach.

5.1 fMRI Data Analysis

Functional Magnetic Resonance Imaging (fMRI) captures volumetric images in a sequence [1]. A complete brain volume can be captured every two seconds. The images can be tuned to accentuate distributions of deoxy- and oxy-hemoglobin, which have been shown to be closely related to neuronal activity, providing a valuable view of the internal workings of the brain. ImageTcl is being used to develop algorithms which detect, localize, and parameterize this activity [8].

fMRI data analysis is a good example of the importance of algorithm development at the compiled level. A three minute fMRI sequence generates over 50 megabytes of image data. The ImageTcl activation localization implementation can process this data set in about two minutes.

5.2 Text-to-speech Alignment

Text-to-speech alignment is the temporal alignment of text to speech audio. The process is described in more detail in other literature [11]. Briefly, text is converted to a biphone directed graph (*Biphones* are sub-phonemic units of pronunciation). Audio is converted to biphone probabilities in ten millisecond time frames. A modified Viterbi algorithm is used to compute the alignment of the two.

This application is unique in that the Oregon Graduate Institute speech tools, another Tcl/Tk based system, were added to ImageTcl. Rather than building these components into the base system, the tools were treated as a plug-in component. A single ImageTcl *build file* was created to include the OGI libraries. This illustrates the flexibility of the ImageTcl automatic build tools: outside components are easy to add to a development project.

5.3 Cut and Pause Detection

The DEVLAB has been very interested in *information retrieval* (IR) in multimedia [5]. One common component of video IR systems is *cut detection* [16]. Cut detection is the location of camera edits in digital video sequences. This information provides a logical segmentation of the content. Each "cut" is a contiguous unit and can often be represented using a single frame called a *key frame* (or a small set of key frames). Several algorithms have been implemented at the DEVLAB.

Cut detection algorithms are often highly parameterized. Nearly all systems have fixed thresholds for detection. Many also have window parameters, histogram bucket counts, etc. Optimization of these parameters and application of test sequences is an on-going process. In ImageTcl the parameters are available at the scripting level and can be easily moved into user interface components.

An alternative video segmentation method is *pause detection* [4]. Pause detection can be described as "inverse" cut detection. Cut detection searches for frame pairs with large movement which cannot be explained by camera motion. Pause detection searches for periods of minimal motion. The DEVLAB has been studying American Sign Language (ASL) video. In this video, sequences do not have conventional camera cuts, but do have inter-element pauses. Reliable detection of these pauses is ongoing research. In addition, the result is a temporal sequence giving the duration and location of pauses. This sequence must be viewed with the original video, often frame-by-frame, in order to gauge algorithm effectiveness.

6 Summary

ImageTcl, a multimedia algorithm development system based on Tcl/Tk, is described. ImageTcl provides the algorithm developer with a fast and effective prototyping environment for new multimedia algorithms. The focus of the system is on algorithm development in C++ and testing and user interface development in Tcl. Numerous system automation tools and library components assist the developer in this process. Several applications of ImageTcl in multimedia research are also described.

7 Acknowledgments

The author wishes to acknowledge Fillia Makedon for project assistance, development facilities and support, and much wisdom. The American Sign Language research is in association with Carol Neidle from Boston University, Benjamin Bahan of Gallaudet University, and Otmar Foelsche of Dartmouth College. Several graduate students who have contributed to ImageTcldevelopment include James Ford, Jim Shain, and Xiaowen Liu. They have also participated in ImageTcl application projects at the DEVLAB.

8 Availability

Information about ImageTcl is available at `http://devlab.dartmouth.edu/imagetcl/`. At this time the system is considered to be in an alpha state and is not generally available, though specific requests will be considered.

References

[1] Peter A. Bandettini and Eric C. Wong. *Echo Planar Imaging*, chapter in Echo-Planer Magnetic Resonance Imaging of Human Brain Activation. Springer-Verlag, 1997.

[2] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[3] Christopher J. Lindblad. A programming system for the dynamic manipulation of temporally sensitive data. MIT/LCS/TR-637, Massachusetts Institute of Technology, 1994.

[4] Xiaowen Liu, Charles B. Owen, and Fillia S. Makedon. Automatic video pause detection filter. Technical Report PCS-TR97-307, Dartmouth College, 1997.

[5] Fillia Makedon and Charles B. Owen. Multimedia data analysis using ImageTcl and applications in automating the analysis of human communication. In *Proceedings of the 3rd Panhellenic Conference with International Participation: Didactics of Mathematics and Informatics in Education*, Patras, Greece, 1997.

[6] James Matthews, Peter Gloor, and Fillia Makedon. VideoScheme: A programmable video editing system for automation and media recognition. In *ACM Multimedia'93*, Anaheim, CA, 1993. ACM Press.

[7] John K. Ousterhout. *Tcl/Tk Engineering Manual.* Sun Microsystems, 1994.

[8] Charles B. Owen. Application of multiple media stream correlation to functional imaging of the brain. In *Proceedings of the International Conference on Vision, Recognition, Action: Neural Models of Mind and Machine*, Boston, MA, 1997. In submission.

[9] Charles B. Owen and Fillia Makedon. Multimedia data analysis using ImageTcl. In *Gesellschaft für Klassifikation e.V.*, University of Potsdam, Potsdam, Germany, 1997.

[10] Charles B. Owen and Fillia Makedon. Multimedia information retrieval development using ImageTcl. In *20th International ACM SIGIR Conference on Research and Development in Information Retrieval*, Philadelphia, PA, 1997. Rejected.

[11] Charles B. Owen and Fillia Makedon. Multiple media stream data analysis. In *Gesellschaft für Klassifikation e.V.*, University of Potsdam, Potsdam, Germany, 1997.

[12] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, second edition, 1993.

[13] Jonathan Swartz and Brian C. Smith. A resolution independent video language. In *ACM Multimedia'95*, pages 179–188, San Francisco, CA, 1995. ACM Press.

[14] The MathWorks, Inc. The MathWorks web site, http://www.mathworks.com/, 1997.

[15] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Inc., Sebastopol, CA, 1991.

[16] Hong Jiang Zhang, Atreyi Kankanhalli, and Stephen W. Smoliar. Automatic partitioning of full-motion video. *Multimedia Systems*, 1:10–28, 1993.

Nsync - A Constraint Based Toolkit for Multimedia[*]

Brian Bailey
Joseph A. Konstan
Department of Computer Science
University of Minnesota
{bailey, konstan}@cs.umn.edu
http://www.cs.umn.edu/Research/GIMME

Abstract

Nsync (pronounced 'in-sync') is a declarative multimedia synchronization toolkit, implemented entirely in Tcl, designed to ease the complexity of designing innovative, interactive multimedia applications. Nsync does not represent a new multimedia synchronization model; rather, it provides a set of building blocks in the form of temporal and non-temporal constraints useful for specifying both the synchronization and interaction properties of an application. Nsync depends only on the logical time system provided by the Berkeley Continuous Media Toolkit, thus allowing any application using a similar notion of time to also benefit from the Nsync constraint mechanism. Nsync presents a new and powerful environment for rapid development of highly interactive multimedia applications.

1 Introduction

A multimedia application can be partitioned along three axes:

- *Content*. The text, graphics, images, audio, or video used within the application.
- *Synchronization*. The temporal ordering of the content.
- *Interaction*. The control the user exercises over both content and synchronization.

As Figure 1 shows, the complexity in building a multimedia application increases as the desire for content, synchronization, and interaction increases. Most multimedia toolkits, such as the Berkeley Continuous Media Toolkit [10], have primarily focused on providing content as opposed to synchronization or interaction. Previous efforts to provide synchronization support for applications have resulted in the development of numerous synchronization models [6, 9, 12]. These models can typically be categorized as:

- *Timeline*. Defines actions such as the starting or stopping of a media object to occur at a specific time.
- *Hierarchic*. Provides two operators, "parallel" and "serial", which can be applied to the endpoints of different media objects.
- *Reference point*. Synchronization points are defined within media objects which may inhibit or cause other media objects to playout.
- *Event based*. Applications express interest in system events, such as the starting or stopping of a media object, and are notified when those events occur.

As expected, each of these models has its own strengths and weaknesses, and none provide a complete set of synchronization abstractions (see [8] for a good overview and comparison of these models). Modeling the interaction properties of an application has received attention from a few systems such as [4], but in general has not been addressed.

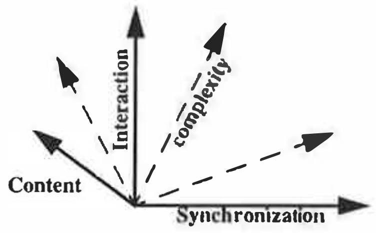


Figure 1: The Content, Interaction, and Synchronization axes of a multimedia application. Complexity is increased by traversing any of the axes outward.

[*] Supported by NSF IRI-94-10470 and a grant from DMRC.

2 Background

Because our work primarily focuses on providing support for the synchronization and interaction aspects of multimedia applications, we decided to use an existing media toolkit to provide the necessary content. The media toolkit chosen was the Continuous Media Toolkit (CMT) developed at the University of California Berkeley by the Plateau project [10]. CMT is a flexible low-level toolkit for building distributed continuous media (CM) applications. CMT provides support for a variety of video and audio formats, transport protocols, and hardware playout devices. Each of the CMT media objects can be dynamically loaded as a Tcl extension. These objects then provide an associated Tcl interface for creating, configuring and removing the object.

CMT also provides a distributed clock object called the Logical Time System (LTS) for media stream control. An LTS is shared by all the CM processes that map real time to "logical" time. Logical time has no start or end and can be thought of as an infinite timeline in both directions. An LTS object maintains three attributes: *value*, *speed* and *offset*. *Value* represents the current value of the clock, which we commonly refer to as *media time*, and which can be set to a new value in order to support random access within a media stream. *Speed* represents the ratio of the *media time* speed to real time. For example, a speed of 2.0 indicates that the media is being played at twice its natural rate and a speed of -1.0 indicates that it is being played in reverse at normal speed. *Offset* is a scalar that is used to complete the mapping between *media time* and real time:

$$\textit{Media-Time} = \textit{Speed} * \textit{Real-Time} + \textit{Offset}.$$

An LTS can be set by setting any two of {*media-time*, *speed*, *offset*}, and the third will be computed. See Figure 2 for LTS code examples.

CMT media objects are attached to an LTS which is used to provide the timing basis for the playout of individual media frames. An LTS, which from now on will be referred to as a clock object, has the following properties:

- Multiple media objects or streams can be attached to a single clock to achieve fine-grained synchronization; i.e., lip-sync quality.
- Multiple clocks can be created and attached to different media streams to support unsynchronized (or differently synchronized) playback, or to support other application objects that also need a notion of time; e.g., animation.
- *Speed* and *offset* do not change during normal playout without intervention by either the user or other system events.

```
(a) set clock [lts ""]
(b) $clock config -speed 1 -value 0
(c) $clock destroy
```

Figure 2: Tcl code examples for (a) creating, (b) configuring, and (c) destroying an LTS.

- Clocks can be perfectly synchronized by setting the speeds and offsets to identical values. Since *media time* is computed from real time, two clocks with the same *speed* and *offset* will always have the same *media time*.
- Clocks always progress in a piecewise linear fashion (see Figure 3 below).

Because of its piecewise linear property, the amount of time it will take a clock with its current *value*, *speed*, and *offset*, to reach a future time instant can be predicted. However, if any of the three attributes of a clock change, then this predicted value will no longer hold. Utilizing the predictable nature of a clock will be shown later, but for now it suffices to realize that a method for clock attribute change notification is needed. Together, the OAT [11] and TclProp [5] systems provide this functionality. OAT, or object attribute trace, provides a protocol for adding traces on new types of Tcl objects; e.g. clocks and their attributes. TclProp further builds on OAT to provide triggers and one-way constraints.

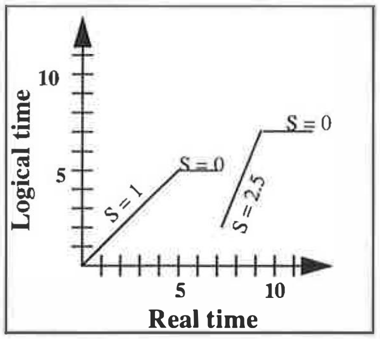


Figure 3: A graph of the CMT clock object. From real time 0 thru 5, logical time progresses at the same speed as real time, signified by speed (S) = 1. From real time 5 thru 7, S = 0, and logical time does not change. At real time 7, random access has been performed by setting the clock's value attribute back to logical time 2, and the speed is also set to 2.5. Logical time now progresses from logical time 2 at 2.5 times the rate of real time. At real time 9, S = 0, and logical time continues unchanged.

3 The Nsync Toolkit

With CMT providing the basic media stream support for applications, we focused our efforts on providing support for both the synchronization and interaction properties. The goal was to develop a simple methodology for defining a broad base of useful relationships involving clock objects (thus controlling the temporal layout of the media) and user interface events; e.g., button presses. This methodology has been embodied in the Nsync (pronounced 'in-sync') synchronization toolkit which has been implemented entirely in Tcl. At the core of Nsync is a declarative constraint language which supports each of the following:

- *Temporal constraint.* A Tcl expression consisting of constants, scalar variables, arithmetic operators, equality or inequality relationships, boolean connectives, and at least one clock object. Temporal constraints are useful for specifying the synchronization aspects of a multimedia application (see Figure 4a).
- *Non-temporal constraint.* A Tcl expression consisting of constants, scalar variables, arithmetic operators, equality or inequality relationships, and boolean connectives. Non-temporal constraints are useful in user interface development and therefore address the interaction component of multimedia applications (see Figure 4b).
- *Combination*s of temporal and non-temporal constraints. Both disjunction (||) and conjunction (&&) can be applied to any combination of temporal or non-temporal constraints. These combinations allow the synchronization and interaction issues to be simultaneously addressed (see Figure 4c).
- *Enforcement action.* A user-defined Tcl command invoked when the constraint expression becomes true (See Figure 4).

Examples of constraint specification in Nsync are given in Figure 4. Each of the constraints are specified using the following syntax:

`When` *expression action*

where `When` is the Tcl procedure name, *expression* is the constraint to be maintained, and *action* is the enforcement action. Semantically, the constraint in Figure 4a states that *whenever* `clock1` *becomes* greater than `clock2` plus the value in the `skew` variable OR *whenever* `clock2` *becomes* greater than `clock1` plus the value in the `skew` variable, set the value of `clock1` equal to the value of `clock2`. Because the attributes of either clock may be changed at any time; e.g., through temporal access controls, the constraint may need to be enforced (by calling the user-defined enforcement action) many times.

Several properties about Nsync constraints are notable:

- *Declarative.* Only the what is specified to the Nsync system without specifying any of the how. In other words, the constraints are described to the system and the system determines when to invoke the corresponding enforcement action.
- *Dynamic.* Nsync constraints can be created, deleted, or modified at run-time. For example, the `skew` variable from Figure 4a can be modified at run-time which will affect when the enforcement action is invoked.
- *Expressive.* Nsync constraints can be used to specify a wide variety of different constraint relationships.
- *Understandable.* Nsync constraints are very intuitive as they can simply be read from left to right. The English equivalent of a constraint can be stated by using the following grammatical template: "whenever *expression* becomes true, perform this *action*."

With the ability to explicitly combine temporal and non-temporal constraints, Nsync directly supports both the synchronization and interaction aspects of multimedia applications.

```
set action [list $clock1 config -value \[$clock2 cget -value \]]
When "($clock1 > $clock2 + \$skew) || ($clock2 > $clock1 + \$skew)" $action
                    (a)
When "(\$skew_desired == 1)" {set skew $current_skew_value}
                    (b)
set action [list $clock1 config -value \[$clock2 cget -value \]]
When "(\$skew_desired == 1) && (($clock1 > $clock2 + \$skew) || ($clock2 > $clock1 + \$skew))"
        $action
                    (c)
```

Figure 4: Tcl code examples for **(a)** Temporal constraint defining skew control **(b)** Non-temporal constraint to retrieve the skew value when desired, signaled by clicking a checkbox, and **(c)** a combination of a temporal and non-temporal constraint to enforce the skew relationship of **(a)**, to be within the limit obtained from **(b)**, but only when desired by the user. Skew control requires the logical time of two clocks to be within a specified value from one another.

4 Implementing Nsync with Tcl

The entire Nsync toolkit has been implemented using the Tcl language. Tcl was chosen as our implementation language for the following reasons:

- *Constraint specification is not time critical.* Most constraints will be specified long before they are actually enforced. Thus, the specification of the constraints is not time critical, however we do recognize that the enforcement of the constraints may be time critical.
- *Dynamic code evaluation.* By using the `subst` and `eval` commands, the Nsync constraint mechanism is very flexible. For instance, any valid Tcl command, whether it is a procedure call or the setting of a variable, can be used as the enforcement action of a specified constraint.
- *Existence of needed tools.* Nsync leverages existing tools such as the Berkeley Continuous Media Toolkit for basic stream support, OAT for object attribute trace support, and TclProp for both object change notification and non-temporal constraint specification.
- *Timed event queues.* Tcl already provides excellent support for timed event queues through the `after` command.
- *User interface components.* Nsync also provides an integrated set of Tk based mega-widgets such as vcr controls, jog shuttle controls, and random access bars.

The Nsync toolkit architecture is shown in Figure 5.

5 Nsync Implementation

Looking at the skew constraint shown in Figure 4a, it may not be obvious why neither the Tcl `eval` mechanism nor TclProp can be used to evaluate the constraint expression. The key reason is the inclusion of *media time*. *Media time* contains several properties that make either approach impractical:

- *Media time* is continuously changing, except when the clock speed is 0. In TclProp, formulas and triggers would need to be re-evaluated continuously which is impossible and impractical.
- *Media time* often should be compared using inequalities (such as whether a particular clock's *media time* is >= another clock's *media time*) which are not supported by TclProp.
- The truth value of the constraint expression is no longer just TRUE or FALSE. An additional value of WILL BECOME TRUE (WBT) is necessary and will be introduced in section 5.2. Obviously, neither the Tcl `eval` mechanism nor TclProp can produce this temporal truth value.

The Nsync implementation consists of four components:

- *Compiler.* Parses the constraint expression and converts it into a postfix expression stack for fast runtime evaluation.
- *Evaluator.* Determines the truth value of the constraint expression. If the expression evaluates to TRUE or WBT, then the Evaluator calls upon the Scheduler to invoke the enforcement action immediately (TRUE) or at the predicted time (WBT).
- *Change Monitor.* Watches the relevant scalar variables and clock attributes used in the constraint expression. If any of these change, the Scheduler is notified.
- *Scheduler.* Schedules the enforcement action to be invoked at the requested time.

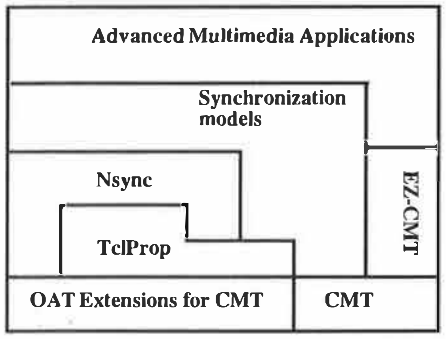


Figure 5: The Nsync toolkit architecture. Nsync leverages existing components such as CMT, OAT, and TclProp to provide synchronization and interaction support.

5.1 Compiler

The Compiler is invoked by calling the `When` command with two parameters. The first parameter represents the constraint expression and is treated as a Tcl string. The second parameter represents the enforcement action and is simply stored for later use by the Scheduler. Within the constraint expression, any attribute of a clock may need to be referenced and some method for clearly identifying the attribute needed to be developed. To address this issue, Nsync defines the notion of a *typed reference*. A typed reference is a string conforming to the following format:

{TYPE object attribute}

When the user requires an LTS attribute in the constraint expression; e.g., *speed*, the string *{LTS ltsname speed}*

Table 1: Truth table for temporal AND

AND	TRUE	FALSE	WBTy
TRUE	TRUE	FALSE	WBTy
FALSE	FALSE	FALSE	FALSE
WBTx	WBTx	FALSE	WBT max (x,y)

Table 2: Truth table for temporal OR

OR	TRUE	FALSE	WBTy
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	WBTy
WBTx	TRUE	WBTx	WBT min(x,y)

is popped from the stack and each of its operands is recursively evaluated. Each operand is evaluated according to the following rules:

- Constants are promoted to a temporary clock object with a speed of zero, and with a value set equal to the constant.
- Scalar variables first have their value retrieved using the Tcl `subst` command. The variable value is then promoted to a temporary clock object in the same manner as constants.
- Arithmetic operators (+, -, *, /) are evaluated by applying the operator to the *value* attributes of the clock objects[*]. Because of constant and scalar variable promotion, each operand of the arithmetic operator is guaranteed to be a clock object.
- The inequality and equality operators (>, >=, <, <=, ==) work similar to the arithmetic operators, except when either of the clock object's speed is not equal to zero (each operand will be a clock object due to previous promotions). When the speed of each clock is zero, the clock object values are compared using the appropriate operator and TRUE or FALSE is returned. However, when one of the speeds is non-zero, then a prediction algorithm is invoked which may return TRUE, FALSE, or WBT along with a predicted time. The following code fragment represents the computation performed for the > operator between two clock objects. This code sample produces the correct result for any combination of actual media clocks and temporary clocks resulting from promotions.

```
set value1 [$clock1 cget -value]
set speed1 [$clock1 cget -speed]
set value2 [$clock2 cget -value]
set speed2 [$clock2 cget -speed]
if {$value1 > $value2} {
    #clock1 value is already > clock2 value
    return TRUE
} elseif {$speed1 <= $speed2} {
    #value1 < value2 and impossible for the
    #two clocks to cross
    return FALSE
} else {
    #value1 < value2, but clocks will cross,
    #predict that future time
    set prediction [expr (($value2 - $value1)
                         / ($speed1 - $speed2))]
    return [list WBT $prediction]
}
```

[*]When applied to two clocks with non-zero speed, the result is undefined.

- The boolean AND (&&) and OR (||) binary operators are summarized by the truth tables given in Tables 1 and 2 above. As Table 1 shows, if both operands currently have the temporal boolean value WBT, the AND operator returns WBT along with the *later* (max) of the two predicted values because that is when *both* of the operands will be true. Similarly, as Table 2 shows, the OR operator will return WBT along with the *sooner* (min) of the two predicted values because that is the time when the *first* of the two operands will be true.

Every time the constraint expression stack is evaluated, the Evaluator applies the above rules to determine the status of the constraint expression. If the constraint expression evaluates to

- FALSE. No action is taken.
- TRUE. The Scheduler is called upon to invoke the enforcement action immediately.
- WBT. The Scheduler is called upon to invoke the enforcement action at the predicted time.

5.3 Change Monitor

As shown in Figure 3, logical time progresses at a pre-

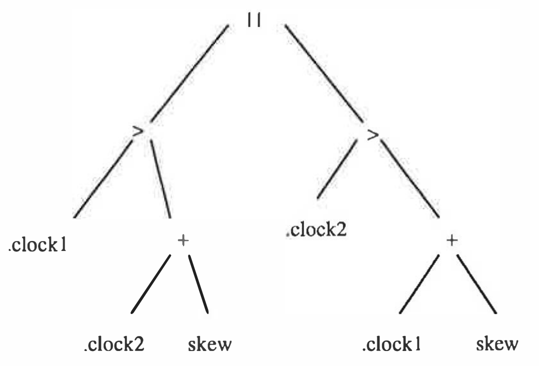


Figure 6a: The parse tree representation of the skew constraint example given in Figure 4a.

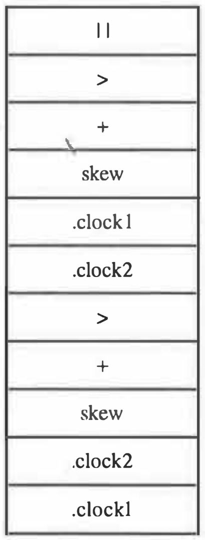


Figure 6b: The postfix expression stack for the parse tree shown in Figure 6a.

is specified[*]. Alternatively, the user may simply invoke an Nsync API which takes a clock variable and attribute, and returns the proper format.

To reduce the need for special quoting, the `When` command assumes all variable substitutions have been made prior to procedure invocation. However, if the user wants to pass in a scalar variable, and not its current value, thus deferring substitution until the constraint is evaluated, then a backslash should be placed before the dollar sign[†].

Once the `When` command is invoked, a lexical analyzer tokenizes the input string (constraint expression) and passes each token to the parser upon demand. The parser uses the tokens to build a parse tree according to the BNF language description[‡]. The parse tree is built using a combination of a Tcl array and several Tcl lists. Each parse tree node contains a value (representing the lexical token) and a list of elements, each of which is an index to another parse tree node. Once the constraint expression is successfully parsed, an equivalent postfix expression stack, used for efficient run-time evaluation, is created. Because of Tcl's built in support for list manipulation, converting the parse tree to an expression stack was extremely simple. See Figure 6a and 6b for the parse tree and stack representation of the skew constraint example in Figure 4a.

[*]The clock variables used in Figure 4 are actually typed references with the attribute being *value*, but are not shown for clarity.

[†]See the examples in Figure 4.

[‡]The BNF is available as part of the Nsync distribution available at http://www.cs.umn.edu/Research/GIMME

5.2 Evaluator

The Evaluator is responsible for determining the truth value of the constraint expression. With the addition of temporal variables, constraint expressions now have *three* possible values:

- FALSE. The constraint expression is currently false, and will stay false at least until a clock's attribute or scalar variable used within the expression is changed.
- TRUE. The constraint expression is currently true, and will stay true at least until a clock's attribute or scalar variable used within the expression is changed.
- WBT. The constraint expression is currently false, but will become true in a predictable amount of time given it current state.

We also recognize that WBT also has an inverse *Will Become False* (abbreviated WBF). However WBF has not been implemented for two reasons:

- The semantic meaning of the `When` command would become ambiguous. Will the enforcement action be invoked when the constraint expression becomes true or when it becomes false?
- Any temporal constraint which needs the notion of WBF can simply invert its relational operator and use WBT.

In order to evaluate the expression stack, the top element dictable rate until one of the attributes, *value*, *speed*, or *offset* is changed. If one these clock attributes or any other variable used within the constraint expression changes, then all expression stacks referencing either of these needs to be re-evaluated. TclProp watches each of the necessary clock attributes and variables, and notifies the Scheduler if any changes occur.

5.4 Scheduler

The Scheduler is responsible for invoking the constraint enforcement action at the future time predicted by the Evaluator. To perform this function, the Scheduler needs to maintain a timed event queue. Fortunately, Tcl already provides the required functionality with the `after` command. The Tcl `after` command arranges for an arbitrary Tcl command to be executed after a specified number of milliseconds have elapsed*. The command returns an identifier which can later be used to cancel the delayed command. To perform its task, the Scheduler simply calls the `after` command with the predicted time and enforcement action as parameters. Once the predicted amount of time has elapsed, the corresponding constraint expression will have just changed from FALSE to TRUE, and the enforcement action should, and will be invoked. However, if the Change Monitor notifies the Scheduler that a dependent clock attribute or variable has changed, then the Scheduler removes all appropriate `after` commands and notifies the Evaluator. The Evaluator then re-evaluates any expression stack referencing the changed clock attribute or variable and the whole process repeats itself.

6 Discussion

Although Tcl is in most respects a very simple language, it has proven capable of building a complex, yet efficient application. In summary, Tcl provided several mechanisms useful for the development of the Nsync multimedia synchronization toolkit:

- *Array and list support.* Used to quickly implement the constraint expression parse tree and equivalent postfix expression stack.
- *Dynamic code evaluation.* Provided great flexibility when specifying the enforcement action of each constraint.
- *Existing tools.* Reusing or extending existing tools such as OAT, TclProp, and CMT greatly increased our productivity.
- *Packages.* The Tcl package mechanism facilitated good source code modularity.
- *Rapid code development.* Nsync was originally implemented in less than 4 weeks of programming effort.
- *Timed event queues.* The Tcl `after` command provided an excellent interface for invoking delayed commands.

*The `after` command does not guarantee any deterministic boundaries on the actual invocation time versus the asked for time.

Nsync has been implemented in about 3,000 lines of Tcl code. The entire source distribution is available at http://www.cs.umn.edu/Research/GIMME

7 References

[1] Blakowski, G., J. Huebel, and U. Langrehr. "Tools for Specifying and Executing Synchronized Multimedia Presentations," *2nd Int'l Workshop on Network and Operating System Support for Digital Audio and Video.* 1991. Heidelberg, Germany.

[2] Blakowski, G. and R. Steinmetz. "A Media Synchronization Survey: Reference Model, Specification, and Case Studies," *IEEE J. Select. Areas Commun.*, Volume 14, no. 1, January 1996.

[3] Borning, A. and R. Duisberg. "Constraint-Based Tools for Building User Interfaces," *ACM Transactions on Graphics*, 5:4, Oct. 1986, pp. 345-374.

[4] Herlocker, J. and J. Konstan. "Tcl Commands as Media in a Distributed Multimedia Toolkit," *Proceedings of the 1995 Tcl/Tk Workshop* (Usenix Association).

[5] Iyengar, S. and J. Konstan. "TclProp: A Data-Propagation Formula Manager for Tcl and Tk," *Proceedings of the 1995 Tcl/Tk Workshop* (Usenix Association).

[6] Little, T.D.C. and A. Ghafoor. "Interval-Based Conceptual Models for Time-Dependent Multimedia Data," *IEEE Transactions on Knowledge and Data Engineering*, 1993. 5(4): p. 551-563.

[7] Ousterhout, J.K. "Tcl and the Tk Toolkit," Addison-Wesley Publishing Company, 1994.

[8] Pazandack, P., "Multimedia Language Constructs and Execution Environments for Next-Generation Interactive Applications," PhD Thesis. University of Minnesota, 1996.

[9] Rothermel, K. and T. Helbig. "Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams," *IEEE J. Select. Areas Commun.*, volume 14, no. 1, January 1996.

[10] Rowe, L. and B. Smith. "A Continuous Media Player," *Network and Operating Systems Support for Digital Audio and Video, Third Int'l Workshop Proceedings.* 1992.

[11] Safonov, A. "Extending Traces with OAT: an Object Attribute Trace package for Tcl/Tk," P*roceedings of the 1997 Tcl/Tk Workshop* (Usenix Association).

[12] Schnepf, J., J. Konstan, and D. Du. "Doing FLIPS: FLexible Interactive Presentation Synchronization," *IEEE J. Select. Areas Commun.*, volume 14, no. 1, January 1996.

Managing Tcl's Namespaces Collaboratively

Don Libes
National Institute of Standards and Technology
Gaithersburg, MD 20899
libes@nist.gov

Abstract

The NIST Identifier Collaboration Service (NICS) is a proposed service to encourage collaboration among researchers and developers when choosing identifiers, far in advance of when it might ordinarily occur. This would support and enhance standards development activities, and development and communications in a variety of fields from software development to system administration. Implementation of this system would provide immediate and significant time and cost-savings to many technology administrators, researchers, developers, and implementors world-wide.

This paper describes the benefits of NICS to the Tcl community. This paper also briefly describes the implementation of NICS which is Tcl-based internally.

Keywords: collaborative namespace management, data element registry, NICS, Tcl

What's an identifier – So what's the problem?

An identifier identifies things. For example, in a programming language, variables are identifiers. You distinguish one variable from another because they have different names. In HTML, tags are identifiers. For example, h1, h2, h3 and so on, are all identifiers. System calls, library names, port numbers are all identifiers.

So what's the problem? The problem is that choosing the right identifier is tricky. I don't mean choosing from among existing identifiers. That's easy. Just read the manual – or the standard. No, the hard part is coming up with new identifiers. It sounds easy but it's hard.

Tcl's Namespace Disaster

Consider Tcl. Tcl provides no scoping of commands [Ousterhout]. Thus, it is important to choose command names which are unique. This is particularly problematic when defining commands that are meant to work with anyone else's command extensions.

Tcl commands are generally short English words or abbreviations. Despite the availability of namespace management extensions such as incr Tcl, command name collisions are common [McLennan]. Ad hoc solutions such as prefixing all commands with a string unique to the extension reduce the problem but do not solve it.

People may suggest that the promised namespace support may rid Tcl of collisions but this isn't quite what will happen. In fact, the namespaces themselves will then need to be managed. This situation of a shared namespace is not unique to Tcl but is common to any rapidly evolving system undergoing development by parties that have no direct means of or desire to collaborate frequently.

NICS provides a general-purpose registry for identifiers. It is designed specifically to be adaptable to different domains. By registering command names and prefixes with this service, it is possible to avoid the types of conflicts mentioned earlier. The remainder of the paper will describe NICS and how it can benefit Tcl with a very low cost to the Tcl development community. Keep in mind that Tcl is only one such application for NICS.

More about NICS

NICS is a web-based collaboration service specifically for identifiers. It provides features that have come to be expected for any web service: world-wide availability and instantaneous access. The service is accessible to the public, yet all information is authenticated – all information is browseable, but all additions or modifications require authentication appropriate for the information. Registration and authentication are totally automated and immediate – from initial contact to full use. All use is free.

NICS is not merely an online registry. Rather, NICS has appropriate and innovative support specifically for identifier collaboration.

- Users may register identifiers well before their use, thereby allowing very early declarations of intent and feedback on ideas, even before initial prototypes or draft standards.
- Identifier descriptions must include their status with respect to use and standards. Placeholders identifiers, deprecated identifiers, and other non-official or de facto identifiers are encouraged if they clarify knowledge needed by other users.
- Users may register classes of identifiers.
- Users may temporarily register multiple identifiers for the same purpose. For example, inter-

nal disputes within standards teams on identifiers should not prevent dissemination and public comment on the possible choices being considered.

- Users may privately or publicly comment on other identifiers. For example, users can suggest better choices or identify potential conflicts. Public comments avoid other multiple users redundantly contacting the original user.
- Information is available instantaneously. There is no delay between the time identifiers and comments are added and when they can be read by other users.
- Identifiers are allowed to conflict. Temporary disagreements show where attention of research should be focused. It is also possible that conflicting identifiers may merely reflect the real world.
- If requested, conflicts and comments are immediately sent to the identifier owner so that they can respond promptly.

Further application to Tcl

The obvious use of NICS for Tcl is to choosing command names. Extension writers could register their extension's command names. Then other extension writers could avoid choosing names that are already in use or are planned for the future. The alternative is to install all other extensions (or read about them) and look for conflicts. Obviously this is not feasible. And it is impossible to learn about future intents this way. In comparison, NICS is trivial to use.

The Tcl development team has preannounced support for namespaces. Although the details are not yet available, it is likely that this work will directly address some of the problems having to do with command name collisions.

However even assuming the command name collision problem disappeared, several other sources of collisions would remain.

- Namespace names – Namespaces themselves will of course have names. These names themselves can have collisions. Although this should be much less of a problem than command names, it is still a problem since extension authors must often pick these names in ignorance of other extensions.

- Package and App_Init names – Collisions occur frequently in this domain. Not surprisingly, many collisions occur because many packages tackle the same area. For example, the Tcl FAQ lists packages from six different authors to generate random numbers, three packages to handle the Berkeley DBlibrary and five packages to support objects. There are literally hundreds of conflicts like this. And there are also conflicts simply because of a poor choice of names, such as names that are too short to be unique or meaningful. In a description of packages, Ousterhout acknowledges the problem and goes on to advise "Choose prefixes that are relatively short (3 to 6 characters)" [Ousterhout]. It is not surprising that people have sometimes reacted by going to the other extreme – choosing lengthy (and sometimes equally unmeaningful) names in an attempt to avoid collisions.

- TCL_XXX values – The completion codes returned by Tcl commands share an unmanaged namespace. There are five predefined codes (TCL_OK, TCL_BREAK, TCL_CONTINUE, etc.). Anything else is application defined. There is no obvious way for extensions to avoid collisions with one another since these numbers are not easily obtained or manipulated. In my own experience with Expect [Libes95], I took the defensive strategy of using relatively large numbers, far, far away from Tcl's choices. This strategy was weak but there does not appear to be any better alternative.

  This is an example of a namespace that could in theory be managed automatically by Tcl. It's trickier than it first appears, since it must account for completion codes explicitly specified in scripts or passed between interpreters in different processes with potentially different completion mappings. Until such a manager is developed, NICS provides a simple solution for avoiding collisions.

- Math function names – Tcl permits the augmentation of function names for evaluation by expr. Like command collisions, function names that conflict simply replace earlier definitions.

- Library names – Everyone wants their library name to be meaningful and short. But after accounting for the "lib" prefix and versioned shared library suffixes such as "5.20.so", that leaves only four characters for portability to file systems with 14 character filenames, making the universe of library names small indeed. But even without these restrictions, most people choose short library names of no more than 6 or 7 characters.

Some of these examples are only part of a much larger problem. For instance, management of Tcl library names is just a subset of library names in general. Consider the system administrator faced with installing software "off the net". The administrator follows all the directions – configuring, compiling, installing – only to be hit at runtime with "ld.so: fatal: can't open file libQtvso.4, errno = 2" or something similar. At this point, a search of the Web reveals three different Qtv libraries!

Program function may require dealing with yet more collisions, apart from those intrinsic to Tcl. For example, programmers must choose port numbers statically and with trepidation. It's easy to tell what port numbers are in use on a host currently. And of course, you should avoid any ports listed in the IETF list of well-known ports [Reynolds]. However, you cannot account for future clashes. If someone else "claims" a port number and their program becomes "popular enough", you lose. Even if it doesn't become popular, anyone attempting to install both packages on their machine faces conflicts.

Traditional approaches to public namespace management

There are common ways of avoiding collisions in shared namespaces:

- Managed database – The Perl Modules List is an example of collision avoidance by explicit human management [Bunce]. Hand-editted by a person, the list maintains all the well-supported Perl modules and provides placeholders for future module names. The advantages are obvious. Yet there are several drawbacks to this approach. The first is that it is expensive. And if the labor is volunteered, then the database becomes dependent on the good graces of that person and whatever they are willing to do and no more. Indeed, the Perl Modules List maintainer specifically refuses to maintain module location information, claiming understandably that it is too much work. Unfortunately, this makes the Modules List frustrating since a module can be listed at times and yet not appear to exist on CPAN (the official Perl archive) [CPAN].
- Published report or standard – I already mentioned the IETF publication and its drawbacks. By comparison, most traditional standards represent even less useful examples of timely information. Part of the problem is that any developing domain necessarily consists of a standard half and a non-standard half. It simply does not make sense to standardize the half of a domain that is in development. Yet practitioners in the field must be knowledgeable of both halves.

  Word of mouth, intuition, and dumb luck – This is the time-honored traditional approach to combining multiple Tcl extensions. Indeed, it is prevalent in many other domains. For example, linkers have no scoping mechanism. When linking together multiple libraries of C code, you have to cross your fingers and pray that subroutine names from different authors don't collide. If you've been given a library from a vendor without source, life can be very unpleasant.

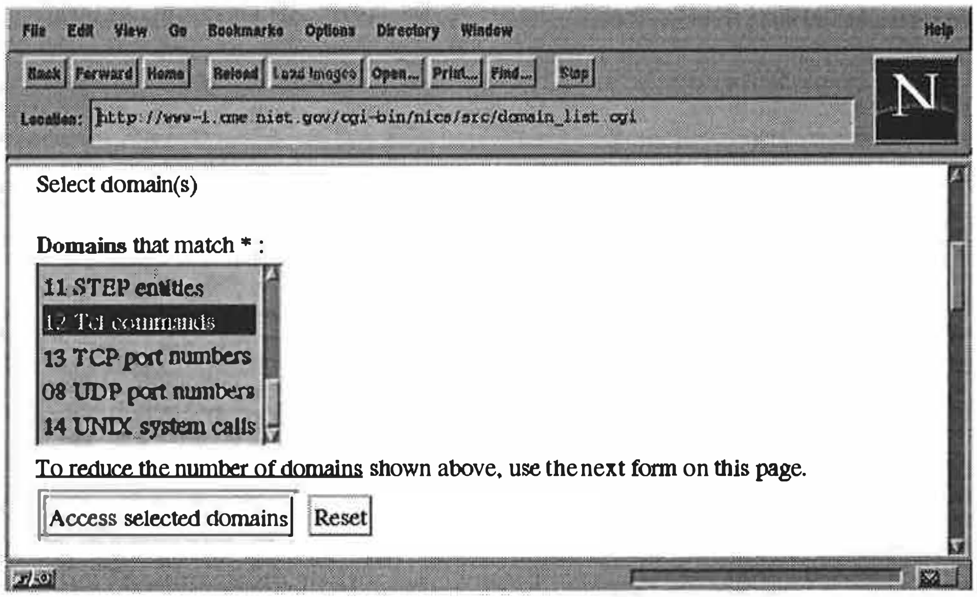
Choosing identifiers is inherently collaborative. The whole point of using identifiers is so that everyone can tell what we are all talking about. If we were just talking to ourselves, we wouldn't need to identify our things. Yet we have no tools to help in this collaboration.

What NICS looks like in use

NICS is a typical web application in the sense that it is CGI-based and uses HTML forms for interactive browsing and input. A file upload facility is available for large updates.

Initially the user selects a domain with which to browse. In figure 1, the user is selecting the domain "Tcl commands". Also shown are some of the other domains likely to be supported by NICS in the future.

Notice that domains do not have version numbers. There is no reason to distinguish, for example, between Tcl 7.4, 7.5, and 7.6 because it is possible to write scripts portable to all of them. Or to put it another way, if there are version-specific differences in the commands such that they could still be accounted for in the current Tcl, the differences should be documented among the commands themselves. Even older versions might not appear in NICS at all. For example, there is no value to listing Tcl 6.0-specific information in its own domain because no one is performing continued development of it or writing extensions for it. Note that historic or dep-


File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Load Images Open... Print... Find... Stop
Location: http://www-i.cme.nist.gov/cgi-bin/nics/src/domain_list.cgi

Select domain(s)

Domains that match * :

11 STEP entities
12 Tcl commands
13 TCP port numbers
08 UDP port numbers
14 UNIX system calls

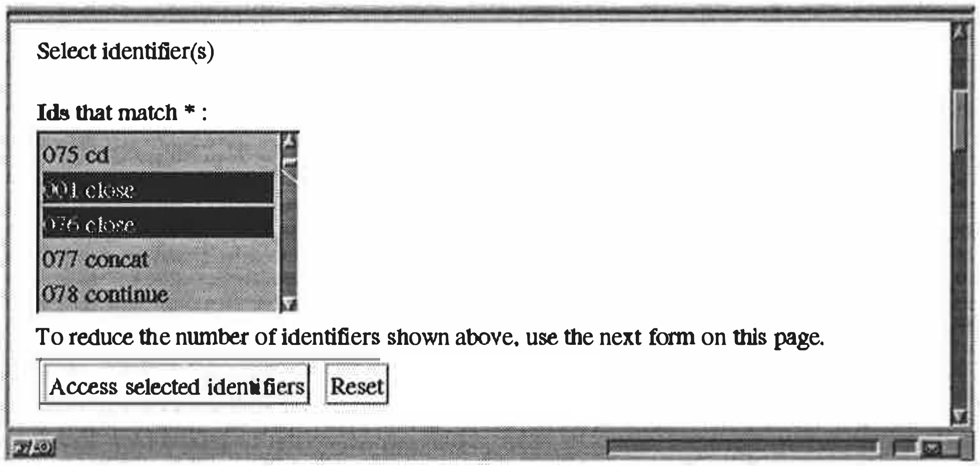
To reduce the number of domains shown above, use the next form on this page.

Access selected domains | Reset

Figure 1: Browsing and selecting domains from the many that NICS manages.

recated commands can be listed (with such a proviso) if there is some useful reason to do so. For example, if an extension chose a name that was defunct but well-known in an earlier release, it might be useful to point this out.

Figure 2 shows a list of some of the identifiers in the domain – in this case Tcl command names. This list alone is useful for someone exploring the domain, perhaps while designing an extension. They can see the kinds of command names people choose, get a sense of the style that is expected, and most significantly avoid collisions with existing command names.



Select identifier(s)

Ids that match * :

075 cd
001 close
076 close
077 concat
078 continue

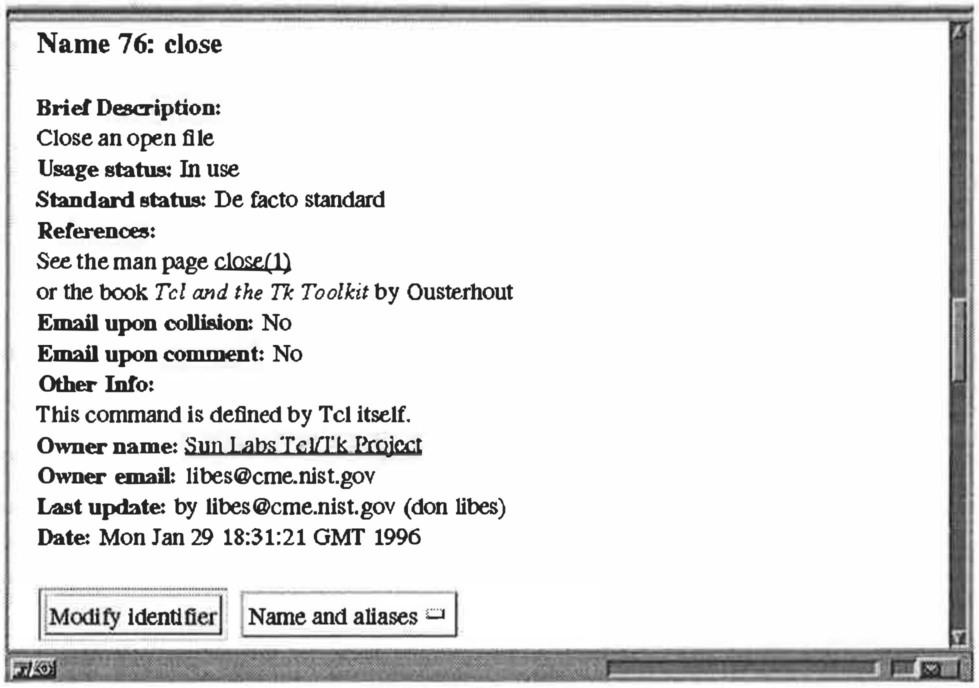
To reduce the number of identifiers shown above, use the next form on this page.

Access selected identifiers | Reset

Figure 2: Selecting two conflicting identifier names from the domain of Tcl command names.

More information than just the name can be accessed by selecting particular identifiers. Here, the command name "close" has been selected – twice. Indeed, one definition is from Tcl and one is from Expect.

Although NICS has tools to discourage collisions, they are not prohibited. Collisions may come about because of any number of reasons – competing proposals, squabbling, or just ignorance. But the reality is, collisions exist in most domains undergoing rapid but decentralized development. Therefore, practitioners in the domain must be aware of these problems – or at least have some easy way of finding out.



Name 76: close

Brief Description:
Close an open file
Usage status: In use
Standard status: De facto standard
References:
See the man page close(1)
or the book *Tcl and the Tk Toolkit* by Ousterhout
Email upon collision: No
Email upon comment: No
Other Info:
This command is defined by Tcl itself.
Owner name: Sun Labs Tcl/Tk Project
Owner email: libes@cme.nist.gov
Last update: by libes@cme.nist.gov (don libes)
Date: Mon Jan 29 18:31:21 GMT 1996

Figure 3: NICS shows the first close definition.

Figure 3 shows detail on the definition for close provided by Tcl itself. Several attributes allow relatively arbitrary text such as the *brief description* and *references.* Since the service runs on the web, it is convenient to take advantage of hyperlinking. For example, the reference section here links to a man page on close that is provided courtesy of some other site. In fact, the textual fields allow arbitrary HTML allowing the possibility of .arbitrary formatting as well as the ability for inline images. In other domains, for example, we have made the descriptions appear to be directly from their standard.

Usage and standard status are also encouraged. Some of the choices for standard status include experimental, de facto, international. It is also possible to indicate "would be a mistake". This is intended to indicate an identifier that should not be standardized – perhaps because it is a placeholder such as unsupported0. This kind of identifier is more common than thought. Often, people don't give much thought to names or functions in a rush to get things working. They may recognize the limited future at the time but others may not and begin embedding such poor names or designs in stone when that was never their intent. Clearly, a name like unsupported0 is an excellent way to indicate the intent of the designer. However, few people are so bold and careful.

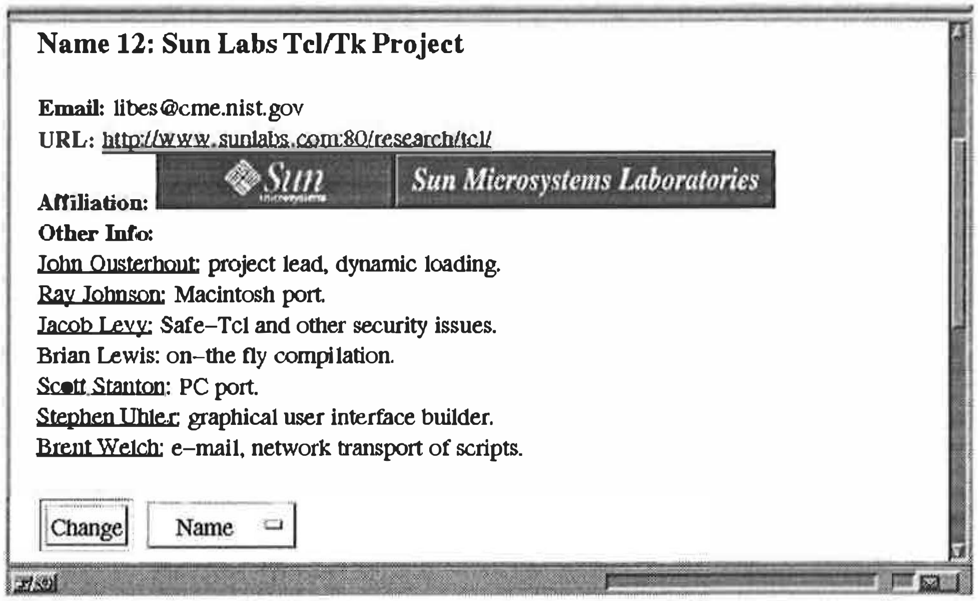
NICS can be told to contact the identifier owner if a collision occurs. This can encourage the two identifier owners to choose different (i.e., better) identifiers – or even better – to work together on the same one. At worst, the two owners will at least be made aware of the problem.

Owners may disable collision notification since such notification is not always appropriate. For example, a domain subset that has been standardized is not likely to be in the position to change.

Anyone can browse this information. However, the information is modifiable only by the identifier owner. In the example shown here, the identifier is a team: the Sun Labs Tcl/Tk Project. An owner may also be a person or any logical entity – as long as it has an email address. The authentication hinges on email reachability [Libes96a]. This technique provides moderate security (appropriate for NICS) with no administration cost to NIST and no special software required by users. Figure 4 shows the owner for the close command.[1]

Figure 5 shows the beginning of the second definition of close. It identifies itself as coming from the Expect extension. The only notable difference is that it declares an alias. Aliases are common in many domains. In the

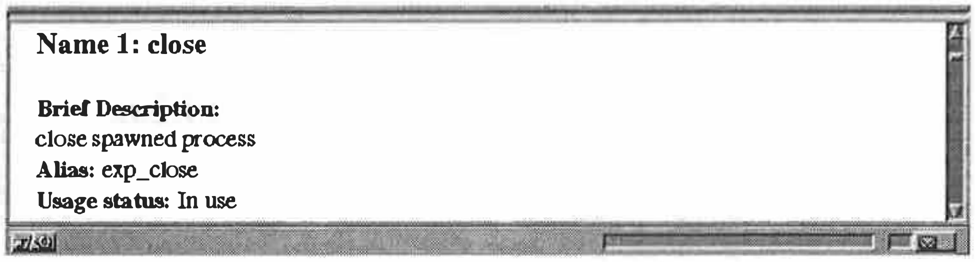
1. Notice that the email authenticates who was actually responsible for adding the information.


Name 12: Sun Labs Tcl/Tk Project

Email: libes@cme.nist.gov
URL: http://www.sunlabs.com:80/research/tcl/
Affiliation:
Other Info:
John Ousterhout: project lead, dynamic loading.
Ray Johnson: Macintosh port.
Jacob Levy: Safe–Tcl and other security issues.
Brian Lewis: on–the fly compilation.
Scott Stanton: PC port.
Stephen Uhler: graphical user interface builder.
Brent Welch: e–mail, network transport of scripts.

Figure 4: NICS shows the owner of the first definition of the close command.

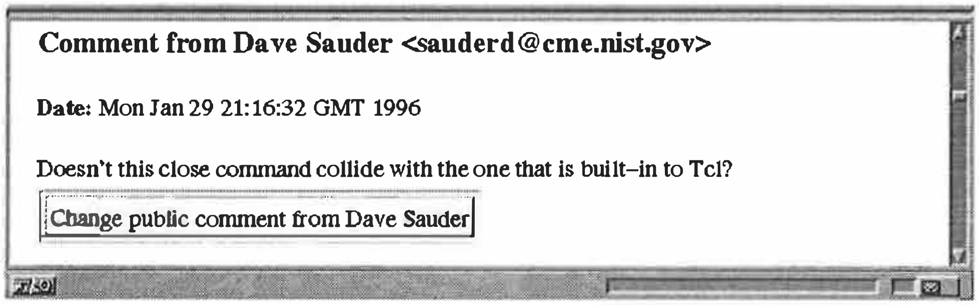
case of Expect, all commands are also aliased with an exp_ prefix. Once informed of aliases, NICS allows queries as if they were full-fledged identifiers themselves but also remembers that they are defined by other identifiers.


Name 1: close

Brief Description:
close spawned process
Alias: exp_close
Usage status: In use

Figure 5: Excerpt of an identifier showing a command name and alias.

At the end of this particular identifier instance is a comment (figure 6). It has been placed by some other user who has observed that Expect's close appears to conflict with Tcl's.


Comment from Dave Sauder <sauderd@cme.nist.gov>

Date: Mon Jan 29 21:16:32 GMT 1996

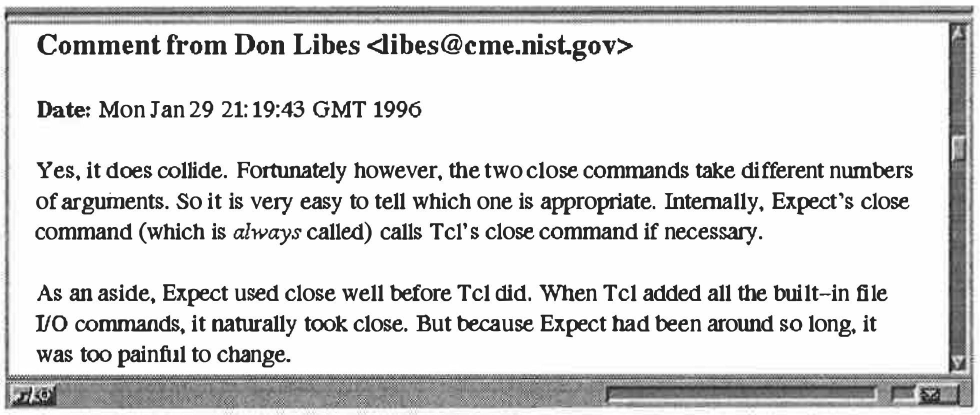
Doesn't this close command collide with the one that is built–in to Tcl?

Figure 6: A public comment attached to an identifier.

Comments in practice tend to be more lengthy (and heated). Comments might also include suggestions for better names, references to both other extensions and people, historical work, future plans, etc. Indeed, it is not uncommon for some people to perform a surprisingly large amount of research in attempting to phrase the observation as carefully and intelligently as possible.

If comments were sent by private email, it is entirely possible that many other people would make the same observations leading to significant wasted effort. Attaching comments to identifiers avoids this. The identifier owner may respond to the comment privately or publicly but may not delete it. Only the comment owner may change or delete it.

Figure 7 shows a possible public response to the earlier comment. This is the type of information that could potentially stop many people from asking similar questions to the earlier one. The response appears after the original comment and can only be changed by the identifier owner.


Comment from Don Libes <libes@cme.nist.gov>

Date: Mon Jan 29 21:19:43 GMT 1996

Yes, it does collide. Fortunately however, the two close commands take different numbers of arguments. So it is very easy to tell which one is appropriate. Internally, Expect's close command (which is *always* called) calls Tcl's close command if necessary.

As an aside, Expect used close well before Tcl did. When Tcl added all the built-in file I/O commands, it naturally took close. But because Expect had been around so long, it was too painful to change.

Figure 7: Owner's response to a public comment.

NICS performs notification of comments in a way similar to that of collisions. For example, identifier owners are notified (if they so request) of comments or changes to them.

Other Features

The web page interface shown here allows identifiers to be added one at a time or en masse using a file upload facility. It is relatively easy, for example to automate population of the commands from an entire Tcl extension. For instance, the Expect commands were added by means of a 35-line script.

NICS has a number of other features. As an example, domains may be *moderated* meaning that the domain owner has control over who can add or comment on identifiers. However these other features are not particularly appropriate to the Tcl community and will not be covered here.

Implementation Notes

The service is written entirely in Tcl. CGI support is provided by the Tcl-based CGI library, cgi.tcl (also entirely written in Tcl) [Libes96b]. Database service is provided by Oracle using OraTcl [Poindexter]. The service runs on a Sun Ultra with access to 55GB Raid storage running Netscape server software. NIST has an SMDS connection (34Mbps) to BBN Planet and a T3 connection (45Mbps) to MCI.[1]

With our current design, over the next three years, we envision on the order of 100,000 registered users, 500 domains, with an average of 10,000 identifiers per domain up to a peak of 1,000,000 identifiers for any one domain. We do not anticipate network bandwidth or space problems during this time. However, if demand outstrips our ability to provide space at a reasonable cost, we would look for and encourage other volunteers institutions to take over specific large domains.

1. Names of companies and products are provided in order to adequately specify procedures and equipment used. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

Duration of the Service

Once development is complete, we envision the service running in an essentially automated mode. Usage and experience must be documented and additional disk space may be required from time to time, but the day-to-day operation will be totally automated. As long as the service is of significant value (i.e., that we claim it will be), we expect to be able to provide the service indefinitely, as the actual expenses (electricity, floor space, disk space, backups, maintenance) are a very small fraction of that of the NIST Information Processing Support Group.

Applying NICS More Generally

Choosing identifiers collaboratively enables easier and faster development of standards and reuse of software. Implementation of this system would provide immediate and significant time and cost-savings to millions of software and standards developers world-wide.

Industry needs the benefits of this service but no single company wants to fund it – this project would benefit many while only the original company pays for it. In contrast, NIST, as a neutral site, is the ideal host for NICS.

Unlike the traditional standards activities performed by NIST, NICS is complementary. In particular, NIST performs research for standards and participates in the creation of standards. More and more, standards are growing increasingly complex. Many standards (STEP, POSIX, etc.) are taking 5 or even 10 years to complete. In part, this is because the technology and in some cases even the standards themselves are being created by many people without collaboration. Without giving up competitive advantages, these people want to collaborate yet they lack the mechanisms to do so in the area that NICS addresses.

Test audiences have identified a large number of domains to which NICS could be immediately applied. We intended on populating NICS with information from a number of these areas to catalyze usage. This is a time consuming task for the very reason that NICS is so useful – because this information is not easily accessible from one place or organized in any uniform way.

In addition to the technical work, crucial non-implementation work is also required. Specifically, various policies must be defined and implemented, and NICS must be thoroughly documented, both for immediate usability and for credibility as a self-sustaining project. Except for research issues, NICS is intended to run by itself as a production service in outlying years. This requires a significant amount of testing and focus on making sure that it is robust.

Status

At the present time, NICS is running in a mode where it is restricted to a small number of domains: the Tcl domains and a few others. This represents an opportunity for the Tcl community – both to manage their namespaces using NICS, and to provide feedback to the usefulness of the concept. We are very interested in suggestions for improvements from the Tcl community which we view as a prototypical audience for the benefit of NICS.

With additional funding, we hope to open the service to other domains in the near future as well as complete other aspects of the service.

Conclusion

NICS is a proposed service to encourage collaboration among researchers and developers when choosing identifiers, far in advance of when it might ordinarily occur. We believe NICS would be of direct benefit to the Tcl community in a variety of ways. While we expect that the one example of command-name conflicts may be addressed by advances in Tcl itself, other namespaces will remain a problem that can be addressed by NICS.

Acknowledgments

Thanks to Mark Williamson, Tu Nguyen, and Jimmy Graham of the NIST Information Processing Support Group for providing web and Oracle service for NICS. Dan Nickchen implemented the first release of the Oracle integration from a prototype file-based version.

This project is financially supported in part by the NIST Advanced Technology Program. Earlier funding was received through a U.S. Department of Commerce Pioneer Grant and a grant from the NIST MEL Director's Reserve.

References

[Bunce] Bunce, Tim, and Konig, Andreas, "The Perl 5 Module List", http://gd.tuwien.ac.at/languages/perl/CPAN/modules/00modlist.long.html.

[CPAN] "CPAN – Comprehensive Perl Archive Network", http://www.perl.com/cpan.

[Reynolds] J. Reynolds, J. Postel, "Assigned Numbers", RFC 1700, USC – ISI, October 1994.

[Libes95] Libes, Don, "Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs", O'Reilly and Associates, January 1995.

[Libes96a] Libes, D., "Authentication by Email Reception", Proceedings of the Fifth System Administration, Networking, and Security Conference (SANS 96), Washington, DC, May 12-18, 1996.

[Libes96b] Libes, D., "Writing CGI Scripts in Tcl", Proceedings of the Fourth Annual Tcl/Tk Workshop '96, Monterey, CA, July 10-13, 1996.

[McLennan] McLennan, Michael, "[incr tcl] – Object-Oriented Programming in Tcl, Proceedings of the 1993 Tcl/Tk Workshop, Berkeley, CA, June 10-11, 1993.

[Ousterhout] Ousterhout, John. K., "Tcl and the Tk Toolkit", Addison-Wesley, 1994.

[Poindexter] Poindexter, Tom, "OraTcl", Tcl/Tk Extensions, ed., Mark Harrison, O'Reilly & Associates, Inc., to appear.

PtTcl: Using Tcl with Pthreads

D. Richard Hipp

Hwaci
6200 Maple Cove Lane
Charlotte, NC 28269

Mike Cruse

CTI, Ltd.
1040 Whipple St. #225
Prescott, AZ 86301

Abstract

Tcl is not thread-safe. If two or more threads attempt to use Tcl at the same time, internal data structures can be corrupted and the program can crash. This is true even if the threads are using separate Tcl interpreters.

PtTcl is a modification to the Tcl core that makes Tcl safe to use with POSIX threads. With PtTcl, each thread can create and use its own Tcl interpreters that will not interfere with interpreters used in other threads. A message-passing mechanism allows Tcl interpreters running in different treads to communicate. However, even with PtTcl, the same interpreter still cannot be accessed by more than one thread.

This paper describes the design, implementation and use of PtTcl.

1 Introduction

Tcl was originally designed for use in single-threaded programs only. But recently, there has been an increasing need to use the power of Tcl in applications that contain multiple threads of control. Unfortunately, the core Tcl library uses several static data structures that can become corrupted if accessed simultaneously by two or more threads. A program crash is the usual result.

There is a least one prior effort to make Tcl thread-safe. Steve Jankowski created a modified version of the Tcl sources called MTtcl [MtTcl] which allows the use of Tcl in a multi-threaded environment. But Jankowski's implementation only works with Solaris threads and on Tcl versions 7.4 and earlier.

This article describes a new implementation of multi-threaded Tcl that is based on POSIX threads [Pthreads] and works with Tcl version 7.6. (An upgrade to Tcl version 8.0 is planned.) We call the package "PtTcl". PtTcl borrows some of Jankowski's ideas but is a completely new implementation.

2 Threading Model

The usual model for a multi-threaded program is that each thread has its own stack used to store subroutine return addresses and local variables. In a compiled program, the hardware in cooperation with the operating system and thread library take care of providing and managing these separate stacks. But in an interpreted language like Tcl, the interpreter must create and manage the separate stacks itself. The standard Tcl interpreter only makes provisions for a single stack. In order to make Tcl truly multi-threaded, it is necessary to change the Tcl core to allow multiple stacks per interpreter.

Unfortunately, the concept of one stack per interpreter is a fundamental assumption in the design of Tcl, and to change this assumption would require a major rewrite of many key routines. In order to avoid excessive rework of the Tcl core, we chose to use a more restrictive thread model for PtTcl.

PtTcl allows an application to have multiple Tcl interpreters running in independent threads, and each thread in a PtTcl program can contain any number of interpreters (including zero). But PtTcl only allows an interpreter to be run from a single thread. If another thread tries to use an interpreter, an error message is returned.

In ordinary Tcl, there is a single event queue used to process all timer and file events. In PtTcl, this concept is extended to one event queue per thread. The fact that each thread has its own event queue is a necessary consequence of the restriction that Tcl interpreters must always be run in the same thread. Recall that the usual action taken when an event arrives is for a Tcl script to run in response. Suppose an interpreter in thread A registers to receive an event, but the event arrives while executing thread B. There is no way for the receiving interpreter, running in thread B, to execute the desired script because that script can only be run from thread A. Hence, if we want to be able to invoke scripts in response to events, each thread must have its own event queue.

Each thread has the concept of a *main interpreter*. The main interpreter is different from other interpreters in the same thread in only one way: you can send messages to the main interpreter.

Messages are a new kind of Tcl event, so a Tcl interpreter running in a given thread will not process any messages until it visits its event loop. A Tcl interpreter visits its event loop whenever it executes one of the standard Tcl commands `vwait` or `update` or one of two commands added by PtTcl: `thread eventloop` and `thread send`.

Messages can be either synchronous (meaning they will wait for a response) or asynchronous (fire and forget). The result returned to the sending thread from a synchronous message is the result of the Tcl script in the receiving thread or an error message if the message could not be sent for some reason. An asynchronous message has no result usually, but it still might return an error if the message could not be sent. Asynchronous messages can be broadcast to all main interpreters or to all main interpreters except the interpreter that is doing the sending.

A message can be sent from any interpreter, not just the main interpreter, or directly from C code. There does not have to be a Tcl interpreter running in a thread in order for that thread to send a message, but a main interpreter is necessary for the message to be received.

Most variables used by a Tcl interpreter are private to that interpreter. But PtTcl implements a mechanism for sharing selected variables between two or more interpreters, even interpreters running in different threads. However, it is not possible to put trace events on shared variables, which limits their usefulness.

Here is a quick summary of the execution model used by PtTcl:

- A single thread can contain any number of Tcl interpreters.
- A particular Tcl interpreter may only be used from within a single thread.
- Each thread has its own event queue.
- A message (in the form of a Tcl script) can be sent to the main Tcl interpreter in any thread.
- Tcl variables may be shared between two or more Tcl interpreters, even interpreters running in separate threads.

3 New Tcl Commands

The PtTcl package implements two new Tcl commands. The "`shared`" command is used to designate variables that are to be shared with other interpreters, and the "`thread`" command is used to create and control threads.

3.1 The "shared" command

The `shared` is similar to the standard `global` command. `Shared` takes one or more arguments which are names of variables that are to be shared by all Tcl interpreters, including interpreters in other threads. Note that both interpreters must execute the `shared` command independently before they will really be using the same variable.

Unfortunately, the `trace` command will not work on shared variables. This is another consequence of the fact that an interpreter can only be used in a single thread. When a trace is set on a variable, a Tcl script is run whenever that variable is read, written or deleted. But, if the trace was set by thread A and the variable is changed by thread B, there is no way for thread B to invoke the trace script in thread A.

3.2 The "thread" command

The `thread` command is more complex than `shared`. `Thread` contains nine separate subcommands used to create new threads, send and receive messages, query the thread database, and so forth. Each is described separately below.

`thread self`

Every thread in PtTcl that contains an interpreter is assigned a unique positive integer Id. This Id is used by other thread commands to designate a message recipient or the target of a join. The `thread self` command returns the Id of the thread that executes the command.

`thread create` [*command*] [`-detach` *boolean*]

New threads can be created using the `thread create` command. The optional argument to this command is a Tcl script that is executed by the new thread. After the specified script is completed, the new thread exits. If no script is specified, the command "`thread eventloop`" is used instead. Assuming the new thread is created successfully, the `thread create` command returns the thread Id of the new thread.

After a thread finishes executing its Tcl script, it normally waits for another thread to join with it and takes its return value. (See the `thread join` command below.) But if the `-detach` option evaluates to true, then the thread will terminate immediately upon finishing its script. A detached thread can never be joined.

Note that the joining and detaching of threads is an abstraction implemented by the PtTcl library. From the point of view of the Pthreads library, all threads created by the `thread create` command run detached.

`thread send` *whom message* [`-async` *boolean*]

Use the `thread send` command to send a message from one thread to another. The arguments to this command specify the target thread and the message to be sent. The message is simply a Tcl script that is executed on the remote thread. The `thread send` command normally waits for the message to complete on the remote thread, then returns the result of the script. But, if the `-async` option is true, the `thread send` will return immediately, not waiting on a reply.

`thread broadcast` *message* [`-sendtoself` *boolean*]

The `thread broadcast` works like `thread send` except that it sends the message to all threads and it always operates asynchronously. Normally, it will not send the message to itself, unless you also specify the `-sendtoself` flag.

`thread update`

This command causes the current thread to process all pending messages, that is, messages that other threads have sent and are waiting for this thread to process. Only thread messages are processed by this command – other kinds of pending events are ignored. If you want to process all pending events including thread messages, use the `update` command from regular Tcl.

`thread eventloop`

This command causes the current thread to go into an infinite loop processing events including incoming messages. This command will not return until the interpreter is destroyed by either an `exit` command or a `interp destroy {}` command.

`thread join` [`-id` *Id*] [`-timeout` *milliseconds*]

The `thread join` command causes the current thread to join with another thread that has completed processing. The return value of this command is the result of the last command executed by the thread that was joined. By default, the first available thread is joined. But you can wait on a particular thread by using the `-id` option.

The calling thread will wait indefinitely for another thread to join unless you specify a timeout value. When a timeout is specified, the `thread join` will return after that timeout regardless of whether or not it has found another thread to join. A timeout of zero (0) can be used if you want to quickly see if any threads are waiting to be joined.

`thread list`

This command returns a list of Tcl thread Id numbers for each existing thread.

```
thread yield
```

Finally, the `thread yield` command causes the current thread to yield its timeslice to some other thread that is ready to run, if any.

4 New C Functions

In addition to the new Tcl commands, PtTcl also provides several new C functions that can be used by C or C++ programs to create and control Tcl interpreters in a multi-threaded environment.

```
int Tcl_ThreadCreate(
   char *cmdText,
   void (*initProc)(Tcl_Interp*,void*),
   void *argPtr
);
```

The `Tcl_ThreadCreate()` function creates a new thread and starts a Tcl interpreter running in that thread. The first argument is the text of a Tcl script that the Tcl interpreter running in the new thread will execute. You can specify `NULL` for this first argument and the Tcl interpreter will execute the command `thread eventloop`. The second argument to `Tcl_ThreadCreate()` is a pointer to a function that can be used to initialize the new Tcl interpreter before it tries to execute its script. The third argument is the second parameter to this initialization function. Either or both of these arguments can be `NULL`. All threads created by `Tcl_ThreadCreate()` are detached.

The `Tcl_ThreadCreate()` returns an integer which is the Tcl thread Id of the new thread it creates. This is exactly the same integer that would have been returned if the thread had been created using the `thread create` Tcl command.

The `Tcl_ThreadCreate()` may be called from a thread that does not itself have a Tcl interpreter. This function allows threads that do not use Tcl to create subthreads that do.

Note that the `(*initProc)()` function might not have executed in the new thread by the time `Tcl_ThreadCreate()` returns, so the calling function should not delete the `argPtr` right away. It is safer to let the `(*initProc)()` take responsibility for cleaning up `argPtr`.

```
int Tcl_ThreadSend(
   int toWhom,
   char **replyPtr,
   char *format,
   ...
);
```

The `Tcl_ThreadSend()` function allows C or C++ code to send a message to the main Tcl interpreter in another thread. The first argument is the Tcl thread Id number (not the pthread_t identifier) of the destination thread. You can specify a destination of zero (0) in order to broadcast a message.

The second parameter is used for the reply. The message response is written into memory obtained from `ckalloc()` and `**replyPtr` is made to point to this memory. If the value of the second parameter is `NULL`, then the message is sent asynchronously. If the first parameter is 0, then the second parameter must be `NULL` or else an error will be returned and no messages will be sent.

The third parameter is a format string in the style of `printf()` that specifies the message that is to be sent. Subsequent arguments are added as needed, exactly as with `printf()`.

The return value from `Tcl_ThreadSend()` is the return value of the call to `Tcl_Eval()` in the destination thread, if this is a synchronous message. For an asynchronous message, the return value is `TCL_OK` unless an error prevents the message from being sent.

```
Tcl_Interp *Tcl_GetThreadInterp(
   Tcl_Interp *interp
);
```

The `Tcl_GetThreadInterp` routine will return a pointer to the main interpreter for the calling thread. If the calling thread does not have a main interpreter, then the interpreter specified as its argument is made the main interpreter. If the argument is `NULL`, then `Tcl_CreateInterp()` is called to create a new Tcl interpreter which becomes the main interpreter. At the conclusion of this function, the calling thread is guaranteed to have a main interpreter and a pointer to that interpreter will be returned.

5 Changes Made To The Tcl Core

The biggest change in PtTcl is the implementation of separate event queues for each thread. In the standard Tcl distribution, the event queue is constructed as a linked list of structures with a static pointer to the head of the list. In PtTcl, we simply converted this static pointer into thread-specific data. Actually, once you get into the details, you find that more than this one static pointer can cause problems. Dozens of static variables in Tcl had to be converted into thread-specific data variables. And, of course, mutexes had to be added to the few static variables that were not converted to thread-specific data.

PtTcl changes the semantics of the `exit` command slightly. In ordinary Tcl, `exit` terminates the whole application, and so it does not worry too much about releasing file descriptors or freeing memory obtained from `ckalloc()`. In PtTcl, `exit` will only terminate the current thread. This necessitated some additional clean-up actions in the Tcl core in order to avoid file-descriptor and memory leaks. Corresponding changes were made to the code that implements the `interp` command in order to get the command

```
    interp delete {}
```

to do the right thing.

The `lsort` command was rewritten to use a merge-sort algorithm [Knuth] instead of the `qsort()` function. This change had the side-effect of making the `lsort` command both recursive and stable. It turns out that it is also a little faster. This change has been folded into the Tcl core as of version 8.0.

In standard Tcl, the `env` array variable contains the values of all environment variables. Changes made to `env` are applied to all interpreters. This behavior is not implemented in PtTcl, however. Each interpreter in PtTcl still has the `env` array containing the environment, but changes to this array are not copied into other interpreters.

During early testing, we discovered that the `printf()` function supplied with MIT Pthreads was not thread-safe. Rather than fix MIT Pthreads, we found it easier to supply our own thread-safe version of the `printf()` function, which we placed in the source file "`generic/tclPrintf.c`". We later used some enhanced features of this alternative `printf()` in the implementation of `Tcl_ThreadSend()`, so even though MIT Pthreads has now been fixed the new `printf()` implementation must remain.

New code was added to implement the **`shared`** and **`thread`** commands. The code for **`shared`** was added to "`generic/tclVar.c`" since it needed access to information local to that file. The **`thread`** command and the new `Tcl_ThreadCreate()` and `Tcl_ThreadSend()` functions are all found in a new source file named "`generic/tclThread.c`".

And, of course, an occasional mutex had to be added here and there, and some functions of the standard C library were changed to their thread-safe equivalents. (Example: `gmtime()` was changed to `gmtime_r()`.) Overall, the implementation of PtTcl was reasonably simple thanks to the extraordinarily clean implementation of the original Tcl core.

6 Obtaining And Building PtTcl

The latest sources to PtTcl can be found at

http://users.vnet.net/drh/pttcl.tar.gz

To build PtTcl, first obtain and unpack the source tree, then `cd` into the directory `pttcl7.6a1/unix` and enter one of the commands

```
./configure --enable-pthreads
```

or

```
./configure --enable-mit-pthreads
```

Use the first form at installations where POSIX threads programs can be built by linking in the special `-lpthreads` library. The second form is for installations that use MIT Pthreads [Provenzano] and require the special `pgcc` C compiler.

After configuring the distribution, type

```
make
```

to build a `tclsh` executable. Note that if you omit the `--enable-pthreads` or `--enable-mit-pthreads` option from the `./configure` command, then the `tclsh` you build will not contain support for Pthreads.

7 Status Of PtTcl Development

We developed and tested PtTcl under the Linux version 2.0. For the Pthreads library, we have used both Chris Provenzano's user-level implementation [Provenzano] (also known as MIT Pthreads) and a kernel-level Pthreads implementation by Xavier Leroy [Xavier] built on the `clone()` system call of Linux. Neither of these Pthreads implementations is without flaw. Under some versions of MIT Pthreads, the `exec` Tcl command did not work reliably. (Later versions of MIT Pthreads work better.) The `exec` command works fine using Linux kernel Pthreads, but under heavy load, the kernel's process table has been known to become corrupt, resulting in a system crash. We suspect that both of these problems are bugs in the underlying Pthreads implementation (or the Linux kernel), not in PtTcl.

PtTcl was written for and has been heavily used in a multi-processor industrial controller that implements its control algorithms using a data-flow model. Each node of the data-flow graph is a PtTcl script running in its own thread. PtTcl has survived extensive abuse testing of the controller software with no errors or memory leaks. But these tests have only exercised those parts of PtTcl which are actually used in the controller application. The `shared` or `exec` commands have not be heavily tested nor have there been many attempts to run more than one interpreter in a thread at a time. We suspect that bugs remain in these areas.

While PtTcl has so far only been tested under Unix, there is nothing in the implementation of PtTcl that would preclude its use under Windows or MacIntosh. All that is needed is a library for the target platform that implements basic Pthreads functionality. We are not aware of any such library but suspect that they do exist. It should not be much trouble to implement Pthreads as a wrapper around the native Windows or MacIntosh thread capability. PtTcl only uses a few of the more basic Pthreads routines, so most of the Pthreads library could remain unimplemented.

8 Acknowledgements

PtTcl was developed for and released by Conservation Through Innovation, Ltd., a manufacturer of environmental and industrial control systems based in Prescott, Arizona.

9 Availability

An online manual and complete source code for PtTcl is available from

http://users.vnet.net/drh/pttcl.html

References

[MtTcl] "MT-Tcl" To appear in *Tcl/Tk Tools* by Mark Harrison. O'Reilly & Associates, Sebastopol, CA. Estimated publication date: June 1997.

[Pthreads] *Portable Operating System Interface (POSIX) - ANSI/IEEE Std 1003.1.* Institute of Electrical and Electronics Engineers, New York, NY. 1996.

[Knuth] Algorithm L on page 165 of *The Art Of Computer Programming* Vol. 3. By Donald E. Knuth. Addison Wesley Publishing Company, Reading, MA. 1973.

[Provenzano] *Pthreads: General Information.* A web page by Christopher Angelo Provenzano. `http://www.mit.edu:8001/people/proven/p-threads.html`

[Xavier] *The Linux Threads Library.* A web page by Xavier Leroy. `http://pauillac.inria.fr/ xleroy/linux-threads/`

A Tcl-based Self-configuring Embedded System Debugger

Dale Parson, Paul Beatty and Bryan Schlieder (dparson@lucent.com)

Bell Labs Innovations for Lucent Technologies

Abstract

The Tcl Environment for Extensible Modeling is a software system from Bell Labs for the simulation, hardware emulation and debugging of heterogeneous multiprocessor embedded systems. These embedded systems contain one or more digital signal processors or microcontrollers that execute real-time software written in assembly language and C. Tcl provides an environment in which embedded system designers can interact easily with their designs. Tcl serves as a processor query language, a modeling language for connecting and scheduling processors, an extension language for adding both model and environment enhancements, and as a user interface implementation language. Tcl's C API and calling conventions provide C and C++-level standards and portable libraries. The Tcl interpreter extends readily into a self-configuring simulation-emulation-debugging tool set. This tool set can use new processor types and new processor arithmetic without tool recompilation. This paper looks at exploitation of Tcl from a system perspective, and at some technical problems and solutions in applying Tcl.

Introduction

Our group in Bell Labs builds software generation and execution tools—compilers, assemblers, linkers, simulators, hardware emulators and debuggers—for a variety of Lucent Technologies embedded digital signal processors (DSPs). Some processors vary at core architectural levels, while other processors differ only with respect to I/O circuitry or memory configuration.

Tcl [1] provides a means for separating processor-specific details from the debugging environment of our simulation and emulation tools. Our Tcl Environment for Extensible Modeling (TEEM) couples a Tcl/Tk-based user interface to a C++ queryable model technology. TEEM configures itself at startup time to support a user-specified set of processors. It queries its processor models to determine processor-specific signals, registers, I/O and memory configurations, and debugger arithmetic semantics.

Tcl finds productive application throughout our environment. From large-scale architectural issues to localized implementation considerations, Tcl provides structure and code that considerably enhances the power and maintainability of our tools.

Section 1 describes where TEEM fits into our system for development of embedded software. Section 2 discusses how Tcl is connected to processor models to support embedded simulation. Section 3 shows how hardware emulation fits into the system. Section 4 discusses the multi-process TEEM graphical interface and our strategy for minimizing application knowledge and communication overhead. Section 5 shows how TEEM may be coupled into third-party environments as a coroutine. Section 6 summarizes our discoveries.

1. Embedded system software generation and execution

Figure 1 is a bird's-eye view of Lucent's Wireless and Multimedia software tools for embedded system programming and debugging. A compiler or assembler translates source code into an object format that a linker binds into an executable file. Compilers for different processors can produce output with different object file formats (COFF, ELF, etc.). Executable file format is one processor-specific system parameter.

The software generation system of Figure 1 is processor-core-specific. TEEM, on the other hand, grows to handle execution of different processor types. TEEM is at the heart of the embedded execution system of Figure 1.

TEEM is *tclsh* (Tcl shell) extended with processor modeling commands from four categories listed below. Tcl procedures can extend each category.

- **Model Management**: The **pssr** command queries available model types, constructs one or more processor instances and deletes model instances. Tcl callbacks that run when a processor is created provide a mechanism for loading and setting up the processor.
- **Model Access**: These commands initialize, examine and modify model instance state (registers, memory, buses and pins). Tcl procedures extend basic query mechanisms to provide information in an application-oriented format.
- **Model Control**: Execution and breakpoint commands drive simulation at the C++ level until breakpoints or exceptions occur. Tcl breakpoint callback procedures can copy data between registers and memory of different processors to model interconnection. Callbacks can also resume processor execution to

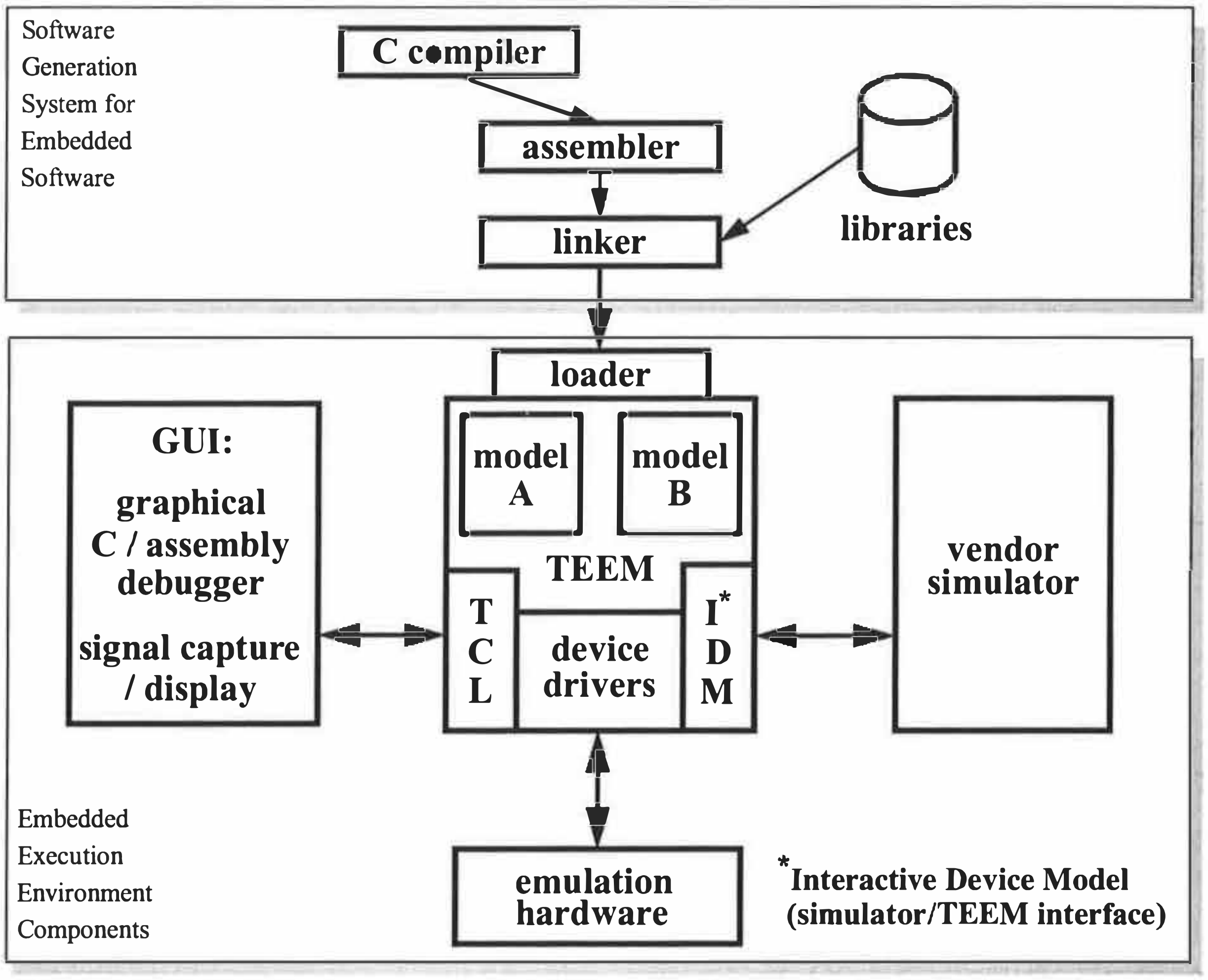

Software Generation System for Embedded Software
C compiler
assembler
linker
libraries
loader
GUI:
graphical C / assembly debugger
signal capture / display
model A
model B
TEEM
TCL
device drivers
I* D M
vendor simulator
emulation hardware
*Interactive Device Model (simulator/TEEM interface)
Embedded Execution Environment Components

Figure 1: Embedded system software generation and execution tools

model scheduling in multiprocessor systems.

- **Model Input / Output**: These commands attach files or Tcl callback procedures to I/O activity. Tcl callbacks support simulation of I/O activity by sourcing and sinking I/O signals between data streams and modeled embedded system I/O.

TEEM operates as a stand-alone tclsh or in conjunction with one or more of the other components shown in Figure 1. In stand-alone mode TEEM supports interaction with an embedded system based on simulation models or emulation hardware. Emulation hardware can include processor evaluation cards or processors embedded in a user's prototype system. When TEEM emulates a processor via hardware, a model instance serves as a database for buffering processor state. Processor execution downloads model state to hardware when execution begins and uploads state to the model upon reaching a breakpoint. Section 3 discusses integration of hardware emulation into TEEM.

TEEM's Tcl command interpreter provides an ideal environment for batch execution, procedural extension and regression testing. However, textual commands give a low-bandwidth user interface, so typical interactive usage requires a graphical user interface (GUI). Instead of including GUI code, TEEM provides a generic socket interface to allow a client application to submit Tcl queries. The GUI process retrieves and updates model state via remote Tcl procedure calls. This optional C / assembly debugger and related graphical signal capture and display tool are discussed in section 4.

In addition to the stand-alone version, TEEM and its GUI may connect to vendor simulators via the IDM interface shown in Figure 1. Typical embedded systems contain analog components and may contain digital components that are not modeled in the TEEM environment. The integration of a vendor simulator such as Model Technology's VHDL circuit simulator or Math Works' Simulink®arithmetic function simulator

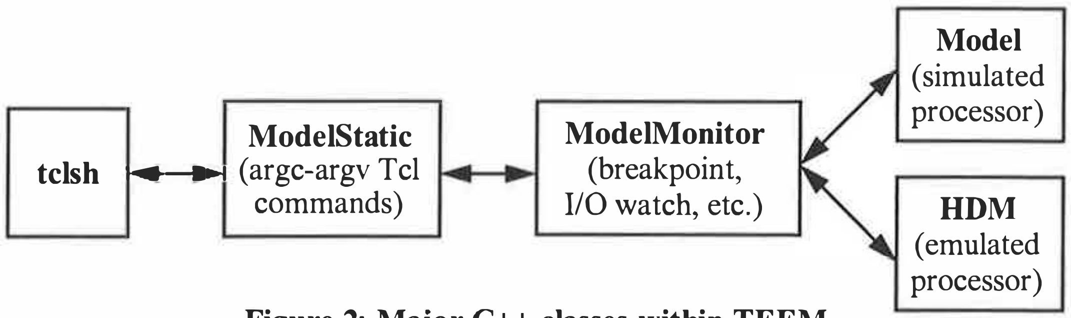


Figure 2: Major C++ classes within TEEM

provides a means for using TEEM models and TEEM debugging functionality in larger systems. While TEEM remains available for processor query and debugging, the external simulator supplants TEEM's built-in model driving functions. Section 5 discusses how TEEM is linked to a host simulator via the simulator's functional API.

2. Tcl Environment for Extensible Modeling = tclsh + processor models

2.1 Tcl-to-model interface classes

Figure 2 shows TEEM's internal structure. Arrows show inter-class communication paths between four C++ classes and the extended *tclsh*. ModelStatic is a Tcl-to-modeling interface class composed strictly of static functions and class-static variables. Each built-in modeling command takes the form of a ModelStatic function that complies with Tcl_CreateCommand's parameter conventions. ModelStatic uses no "this" object or inheritance. Its class variables manage the set of models.

Model is a processor modeling base class that runs simulated circuitry. Each Model-derived class adds processor-specific objects such as registers and I/O ports for storing state. It also adds simulation code for advancing processor state on clock transitions.

The hardware development mode (HDM) class provides access to a real, running processor. It retrieves processor-dependent configuration information from its Model. An HDM object also uses its Model object as a database that stores its processor state information.

Each Model object and HDM object ties to one ModelMonitor object. ModelMonitor is a base class with one derived class per processor core. ModelMonitor performs debugging-oriented monitor tasks that are not part of processor simulation or emulation. Key ModelMonitor jobs include stepping the processor, watching for breakpoint conditions in the Model, directing return to Tcl on uncaught exceptions, and directing callbacks to Tcl on caught exceptions. Both calls forward from tclsh to a ModelMonitor instance and callbacks from ModelMonitor to tclsh travel via class-static functions in ModelStatic.

2.2 Model management

Section 1 listed four categories of ModelStatic commands that manage a set of processors, query/update a processor, run a processor and simulate processor I/O. An important goal of TEEM is to enable debugging of multiple processor instances. Tcl's ability to support object-action command structure enables almost seamless transition of scripts from single to multiple processor debug environments.

The **pssr** model management command can create a processor instance (**pssr new** *modelName*), delete a processor instance (**pssr delete** *instanceName*), and query the set of available processors (**pssr modelTypes**).

Pssr gives access to a dynamically linked library of ModelMonitor C++ class constructors. Each constructed object returns a unique *instanceName* to Tcl. The *instanceName* becomes a Tcl command until the object is destroyed. When *instanceName* is called as a Tcl function, it saves a ModelStatic pointer to the *current processor instance* ModelMonitor object on the C++ stack, copies its own *instanceName*'s address into this *current instance* pointer, passes the remainder of the command to Tcl_Eval(), and restores the previous *current instance* pointer from the C++ stack on return. The next section explores the utility of this dynamic *current instance* pointer.

2.3 Model access

2.3.1 Fxpr: processor-oriented expressions

Assume **pssr new mydsp** creates model instances named "p1" and "p2" on two successive calls. TEEM changes *current instance* twice to interpret the expression:

```
expr "[p1 fxpr r0] == [p2 fxpr r1]"
```

First TEEM sets current instance to p1 and calls ModelStatic's **fxpr** with register name r0 to retrieve the r0 value from the *first* instance of mydsp. Next TEEM sets current instance to p2 and calls fxpr with register name r1 to retrieve the r1 value from the *second* instance of mydsp. Finally, the results of both fxpr queries pass to Tcl's expr to compare the results.

Fxpr accesses a target ModelMonitor using ModelStatic's current instance pointer. Fxpr is syntactically compatible with expr, but its arithmetic semantics are determined by the current processor's ModelMonitor. For example a fixed-point digital signal processor supplies fixed-point semantics, while a floating-point microcontroller supplies floating-point semantics. Fxpr builds and evaluates a processor-neutral parse tree. Parse tree evaluation consults virtual functions in the current ModelMonitor to evaluate numeric constants, model state references (e.g., register r0), variables and arithmetic operations.

Fxpr's use of the current instance pointer typifies the remainder of ModelStatic functions. The processor query, processor execution and I/O attachment commands interact with a model instance whose identity is determined from the current instance pointer. Therefore any remaining ModelStatic function can be prefixed by an *instanceName*. Entering *instanceName* without a command suffix makes *instanceName* the default current instance for all commands that lack an *instanceName* prefix.

Tcl programmers can write processor-neutral model manipulation scripts that do not specify *instanceName*. Their procedures operate on the current processor, using its model-specified semantics transparently. Alternatively, these programmers can prefix specific commands with an *instanceName*. Application of *instanceName* has stacked, dynamic extent. A major advantage of choosing this object-action convention over the alternative action-object convention [reference 1, section 28.3] is that scripts and adjunct tools can prefix an entire set of commands with object *instanceName*. Tcl *evals* the remainder of a prefixed command within *instanceName*'s scope as part of the *instanceName* command. The trailing commands affect the intended processor even though they contain no processor-specification code. The action-object convention would require each Tcl action command to specify its processor, making multi-processor generalization of Tcl commands very difficult to achieve.

Fxpr syntax is a superset of expr. Fxpr adds an assignment operator for copying a value into model state. Model state includes registers, I/O ports and other model signals housed in Model, as well as user-declared signal variables housed in ModelMonitor. Fxpr also has memory vector operations for loading and copying sequences of model memory contents within and between processor Models.

Fxpr provides partial compatibility with an earlier, non-Tcl command line debugger. Users are accustomed to typing expressions such as "r0 = r1 + 3" into the debugger. We did not wish to require that "fxpr" be prefixed to every register/signal access and update (e.g., "fxpr r0 = r1 + 3"), so fxpr is linked to Tcl's **unknown** command. Unknown procedures (such as "r0") call the fxpr parser to determine if the leading token is a symbol for a unit of model state in the current instance. If a valid symbol is recognized, the entire command passes to the fxpr parser, otherwise Tcl's default unknown handler is called. Tcl scripts typically make the "fxpr" prefix explicit to speed processing and eliminate any potential ambiguity.

2.3.2 Other query / update commands

There are several other query commands. The **?** command includes options for determining the names, types, and properties of user-accessible elements within a model. Example names are "r0," "pc," "time." Example types are register, I/O port, memory block, user-declared signal variable. Example properties are signal width and memory width, allocation size and base address. The **width** command retrieves signal and memory width. The **alloc** command includes options for determining embedded memory configuration and for allocating memory for simulation or emulation. A user or TEEM client process can use **?** to find out what objects are inside a processor model. The user or client can then use **fxpr**, **width** and **alloc** to retrieve and update the state of those objects.

Each user command that creates a unit of debugger state returns an instance-handle string to Tcl that identifies that unit of state. Name-to-element bindings are unique within a Model. Each object of class Model includes a C++ symbol table that binds each Model-unique name to an element's type and its defining object within the Model. Each ModelMonitor object houses several symbol tables for administering breakpoints, exceptions, I/O connections, and other user-defined debugging state. In addition to Model state, all ModelMonitor debugger state—the state of breakpoint triggers, user-installed breakpoint and exception callbacks, and the state of Model I/O monitors—are available for Tcl-based query. The accessibility of declarative information about Model and debugger state combines with the interpretive nature of Tcl to support very powerful methods for extension and customization.

Tcl's hash tables and string library functions make provision of platform-independent symbol table objects a simple exercise. TEEM runs on several UNIX® platforms as well as Win32™. At the C++ level TEEM uses only ANSI C libraries and the Tcl library to achieve portability. A considerable portion of the queryable model infrastructure works on top of the Tcl C library.

Query commands determine both identity and content of state-bearing elements in a processor. Query results return Tcl strings or lists. ModelStatic command functions, Tcl extensions, and ancillary tools that interact with models can avoid hard-coded processor specifics.

The **mload** command demonstrates processor independence. Mload loads model memory from an executable file. Recall from Section 1 that different processor cores use different executable file formats. Different compilers or assemblers may pass different debugging information. Within the execution environment ModelMonitor integrates over these variations in file structure. Each core-specific ModelMonitor codes for its processor's executable file structure in a virtual mload helper function. The mload command loads Model memory and ModelMonitor debugging tables for any TEEM processor.

Processor-neutral syntax and processor-interpreted semantics extend naturally to C language expression handling for debugging. We are investigating a model-directed approach to C debugging comparable to the processor-independent techniques of ldb [2] and cdb [3]. These debuggers use processor-independent symbol table information compiled into the executable program to achieve processor independence. A TEEM-based approach houses similar symbol table information in its queryable models and ModelMonitor loaders.

2.4 Model control

ModelStatic control commands start and stop execution. Tcl callback procedures enable multiprocessor scheduling.

The **reset** command resets the current processor's state, and the **step [n]** and **resume** commands advance its state. Model-ModelMonitor pairs cooperate in advancing processor state and monitoring breakpoint and exception status. Step or resume returns processor halt status to Tcl only when an uncaught breakpoint or exception arises in a processor.

The user can set breakpoints on program locations, assorted program-memory interactions, or on the successful, non-zero evaluation of any fxpr expression within a model. Some processor models may augment the default set of breakpoint types, and hardware emulation may restrict available types of breakpoints. A Model may also assert a variety of exception conditions of four severities—note, warning, error and fatal. The list of exceptions is available for query from Tcl. Default processing of a breakpoint successfully returns a breakpoint identifier string (and TCL_OK) to Tcl. Default exception processing prints messages via **puts**, and errors and fatal exceptions return failure diagnostics (and TCL_ERROR) to Tcl.

A Tcl callback procedure name may be supplied as a handler for a breakpoint or exception. An empty string signifying "ignore" may be supplied for any non-fatal exception. When a handled breakpoint or exception arises during processor execution, ModelMonitor calls Tcl_Eval() (via ModelStatic), passing the Tcl callback procedure name and event-identifying parameters that the callback uses as event keys. The current instance is set to the interrupted processor, and the callback procedure has access to the full range of Tcl commands for passing information between processors and reading and writing files and other data streams. If the callback does not call **step** or **resume**, ModelMonitor returns to Tcl upon completion. If the callback does call **step** or **resume**, ModelMonitor resumes execution of the processor after the callback completes. Only fatal errors force a break.

Breakpoints and exceptions can trigger user-defined extensions to a processor model, transparent simulation of processor I/O events, processor state logging and multiprocessor synchronization. A processor scheduler can be as simple as the following Tcl loop:

```
proc sched {pssrlist} {
    # pssrlist is a list of processor instances
    while 1 {
        foreach p $pssrlist {
            $p resume } } }
```

Sched schedules simulation of the processors named in $pssrlist in round-robin order. Each processor resumes execution until it reaches a breakpoint, at which time *sched* schedules the next processor. Within the action of "$p resume" a breakpoint handler can pass information between processors; the handler has the option of resuming its processor without letting control return out to *sched*.

As written above *sched*'s outer loop iterates until an error occurs within one of the "$p resume" actions. Any such error propagates TCL_ERROR out of *sched* in the normal Tcl manner.

2.5 Model I/O

Model and ModelMonitor support attachment of any Model I/O port to a text file. Alternatively Model-level I/O operations may call a user-specified Tcl procedure. A processor input action from an I/O port can cause a call to a Tcl procedure that returns a value for that port. A processor output action to an I/O port can cause a call to a Tcl procedure that takes the output value as an argument. The combination of breakpoint, exception and I/O event callback procedures allows the user to design fully customized, event-driven, multiprocessor simulations within Tcl. The execution of the callback procedures is encapsulated as part of Model execution.

In fact, a processor designer can prototype a model in Tcl by writing only three C++ virtual Model functions as callbacks to Tcl. We normally write final processor Models in C++ for speed, but we have implemented partial Models using Tcl to allow concurrent engineering of the Models. For example, we have written a partial Model that houses only memory in order to test the loader of its companion ModelMonitor class. Unrelated Model functions are stubbed out by binding them to Tcl callback procedures. Using Tcl callbacks as stubs to support incremental construction of partial Models has been very useful.

3. Emulation access to processor hardware

The HDM class shown in Figure 2 provides communications between TEEM and each target processor through an IEEE 1149.1-compliant "JTAG" serial interface. HDM control of JTAG goes through a PC ISA bus card; optional networked access is via a TCP/IP socket interface. The HDM class hides physical communication details by providing a set of generic block transfer functions such as memory upload/ download, register upload/download, processor reset, step and resume, and event monitoring. Plans for more sophisticated controller-based PCI and PC Card interfaces require that physical implementation details be encapsulated in the HDM class hierarchy.

Tcl-queryable models again play an important role. An HDM object determines processor details by querying its Model at startup. With emulation enabled, each HDM object intercepts calls to its corresponding Model, delegating non-emulation work back to the Model. Tcl continues to interact with processor state by querying and updating a Model. HDM uses the Model as a database for storing processor state.

4. Relational Tk mega-widgets and database-event-driven graphical update

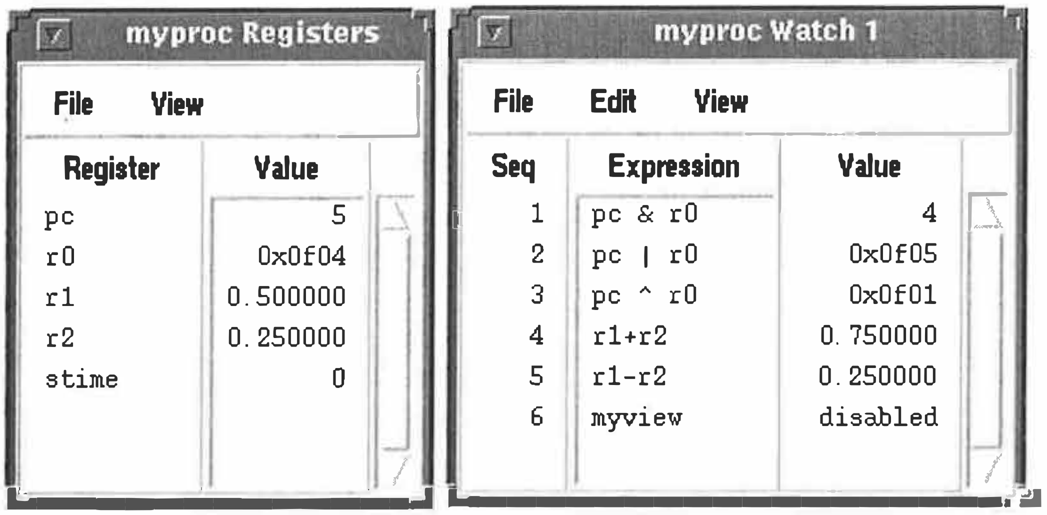
The graphical C / assembly debugger of Figure 1 is a good demonstration of the expressive power of Tcl/Tk. In about 5000 lines of Tcl code it replaced a C-based approach to an earlier debugger that had about 45,000 lines of C GUI code. The Tcl/Tk GUI has far greater functionality thanks to Tcl as a query language. The Tcl/ Tk debugger has a processor source window and command line history and search processing that use straightforward application of Tcl programming and Tk library widgets. The most significant custom savings for Tk came about through the construction of a relational mega-widget that leverages the queryable nature of TEEM. Any queryable data set in TEEM—i.e., the entire model and debugger state—can be displayed in the familiar row-column format of a relational database. We constructed an instantiable relational mega-widget in about 1300 lines of Tcl/Tk. The mega-widget's constructor includes parameters for domain names and associated properties, including formatting and editing callback procedures. User editing of fields and complete tuples is parameter-driven. Mega-widget instances were useful in building display / interaction windows for the following TEEM element types: registers, pins / buses, signal variables, breakpoints, memory vectors, I/O connections, and a spreadsheet-like watch window that can use user Tcl procedures to create customized views into model data sets. Figure 3 gives an abbreviated look at a register window and a relational watch window.

Performance was an issue of concern at the outset of GUI design, especially for PCs. Our solution was to minimize GUI-TEEM communication by doing as much work as possible within TEEM. Tcl "watchdog" procedures query active models periodically within the TEEM process. These procedures send display update messages to the GUI process only when widget contents need to change. This approach eliminates system call and inter-process communication overhead that would be required by GUI-driven model query. This approach makes good use of the interpreted nature of Tcl. No knowledge of the GUI client process is coded into the TEEM server process, but the GUI can source client Tcl "watchdog" code into the TEEM process for efficiency. Efficiency is gained without a loss in modularity.

5. Host simulators and Tcl as a coroutine

5.1 Simulator / TEEM interface

An embedded system designer may wish to model and debug one or more of our processors in a vendor simulation environment in order to gain access to tool capabilities, circuits or arithmetic functions modeled in that environment. A typical simulator architecture requires the user to start a simulator process and specify models as data. The simulator then loads models from



Register	Value
pc	5
r0	0x0f04
r1	0.500000
r2	0.250000
stime	0

Seq	Expression	Value
1	pc & r0	4
2	pc | r0	0x0f05
3	pc ^ r0	0x0f01
4	r1+r2	0.750000
5	r1-r2	0.250000
6	myview	disabled

Figure 3: Relational watch window gives spreadsheet-like views into a model

object files via a dynamic, incremental loader. The models must supply certain access functions specified by the simulator as part of its C application procedural interface (API). The models may call C simulator functions that are also part of the C API.

The Interactive Device Model (IDM) interface of Figure 1 is part of ModelMonitor of Figure 2. It achieves a great deal of leverage from both Tcl and the queryable nature of Model. When a vendor simulator initializes its first TEEM Model after the incremental load of TEEM, ModelMonitor starts tclsh and constructs the communication paths of Figure 2. Thereafter ModelMonitor knits each additional Model into tclsh.

Previous IDM design required a great deal of processor-specific hand code in order to integrate a new processor into a given simulator. At about 800 lines of code per processor variant x 5 variants per year x 2 simulators, we were averaging around 8000 lines of processor-specific IDM code per year, and were limited in the number of additional processors and simulators we could support. With TEEM the IDM can query both the models and the simulator environments, and the number of lines for new processor variants drops to a constant 0. Any processor modeled in TEEM is immediately available to the IDM interface. A new simulator takes a constant of about 100 lines to integrate with TEEM.

5.2 Tcl as a coroutine

Figure 4 illustrates the control problem we encountered when attempting to integrate Tcl as an incrementally loaded subroutine beneath a host simulator. Arrows represent subroutine calls. The first call to a TEEM model initializer calls Tcl_Eval as part of tclsh. Suppose in the process of initializing TEEM, Tcl_Eval sources a commands file that calls Model **step** or **resume**. The external simulator requires that a model initializer or evaluation function must return to the simulator to advance the simulation, i.e., to bring about the **step** or **resume**. Unfortunately if Tcl_Eval returns, the context in which Tcl was sourcing its file will be lost. Tcl cannot simply return to the host simulator.

In this environment Tcl must run not as a subroutine, but as a coroutine peer of the host simulator. With a coroutine organization the Tcl thread has its own execution stack. During Tcl execution this thread is active and the simulator thread is blocked. The Tcl thread can access Models and ModelMonitors, performing queries and other model interactions. Only when Tcl calls **step** or **resume** is it necessary to block the Tcl thread and resume the simulator thread. Step and resume require coroutine resumption logic when run under a host simulator.

Lightweight threads under UNIX® and Win32™ should be able to supply the coroutine mechanism. Coroutines implemented using lightweight threads do not violate Tcl's single-thread limitation, since only one thread is active within Tcl at any given time. TEEM initialization starts a second thread for tclsh and yields control to that thread. ModelStatic's step / resume manages thread execution; synchronization code blocks the simulator thread in order to return from **step** or **resume** to Tcl, and it blocks Tcl in order to initiate a **step** or **resume** within

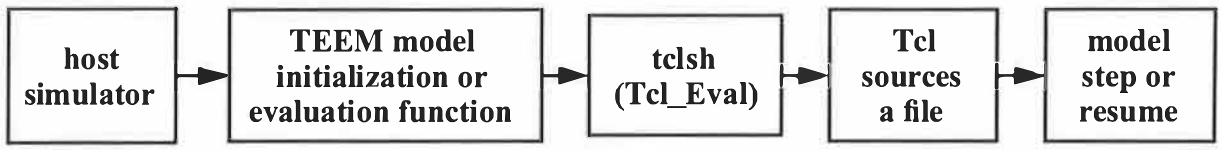


Figure 4: External simulator to model calling stack

the simulator. This scheme worked fine with Tcl and some toy applications, and it worked with Model Technology's VHDL simulator. Unfortunately another simulator not named in this report had mysterious failures when we used lightweight threads. Because we have no means to debug this simulator-specific problem, we switched to a safer but more cumbersome multiprocess approach to coroutines. The blocking threads remain a viable option for cooperative applications, but the external simulator did not cooperate with us.

Our multiprocess solution runs the host simulator and tclsh in separate process spaces. It distributes the class-static functions of class ModelStatic of Figure 2 over an inter-process communication (IPC) medium, and places ModelStatic's data in the appropriate processes. These functions all use the Tcl_CreateCommand-compliant argc-argv calling convention. Consequently distributing these functions across IPC required engineering only one distributed function. The tclsh process houses the main Tcl interpreter, but the simulator process also houses a Tcl interpreter used for decoding and calling the correct ModelStatic functions received via IPC. All user-initiated Tcl interpretation occurs in tclsh, and all ModelMonitor and Model objects reside in the simulator process. Tcl callbacks reverse the order of IPC remote call, calling from ModelMonitor to tclsh. We use an IPC-medium-independent C++ class we already had in hand to transport the remote calls and data, binding it to sockets.

6. Conclusion

Tcl plays many important roles in the TEEM processor modeling environment: 1) Tcl is the model query language that the debugger and ancillary tools use to retrieve processor configuration data. 2) Tcl is a modeling language that supports interconnection of processor instances and prototyping of connecting circuitry. 3) Tcl supports model extension and error handling through breakpoint-triggered, exception-triggered and I/O-triggered callbacks. 4) The Tcl C API and calling convention provide the local and remote procedure calling standard for C and C++-based system components. 5) The Tcl C library provides portable library functions for all required machine/OS platforms. 6) Tcl's well-defined representation of success and error status and return value from a procedure extends readily to a clear notion of a transaction between a tool and a queryable model. Delimited transactions achieve tool synchronization. 7) Each queryable model connects easily to a set of Tk relational mega-widgets that reflect model contents in tabular form. Event-driven widget update, where the change of a widget-displayed datum constitutes an event, minimizes inter-process communication overhead between the modeling and graphical debugger processes. Tcl as the query language supports efficient model event detection in the modeling process. 8) Tcl can integrate as a coroutine into external simulators, providing TEEM capabilities in many contexts.

7. References

1. John K. Ousterhout, *Tcl and the Tk Toolkit*. Reading, Ma.: Addison-Wesley, 1994.

2. Norman Ramsey and David R. Hanson, "A Retargetable Debugger," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 27(7), 22-31 (1992).

3. David R. Hanson and Mukund Raghavachari, "A Machine-Independent Debugger," *Software—Practice and Experience*, Vol. 26(11), 1277-1299 (November, 1996).

GeNMSim - The Agent Simulator

Tcl Based Agent Simulation Software

Ilana Gani-Naor (ilana@milestone.co.il)
Ehud (Udi) Margolin (udi@milestone.co.il)
Raz Rafaeli (raz@milestone.co.il)
Milestone Software & Systems, P.O.B 152, Yoqneam Ilit, Israel

Abstract

Network Management (NMS) application vendors, often encounter a situation where the device being managed or tested (which includes an SNMP agent) is not available at the time of the NMS application development, whereupon this becomes the critical path in the development cycle of the new device. To shorten this critical path, we've developed GeNMSim, which is a Tcl/Tk based Multi Platform SNMP agent simulator. The main features of GeNMSim are user customisation using Tcl callbacks, Portability across Unix and Window95/NT platforms and automatic creation of the simulator database by a set of Tcl based tools. GeNMSim is a commercial product targeted at Data Communication and Telecommunication companies involved in SNMP development.

Introduction

GeNMSim is a Tcl based SNMP agent simulation software. This article gives a short background on the need for this product and looks at the incentive to build this product with Tcl. Then we'll dive a little deeper into the technical aspects of GeNMSim, give some examples of the database structure and cover the process of creating this database, see how GeNMSim runtime is built (with a very short glimpse into the GeNMS technology). The Tcl based callback mechanism follows and the conclusions section discusses the performance and portability problems encountered and the need for better development tools that arised in the process of developing this product.

What is an Agent Simulator

Network Management Systems (NMS)

Network Management Systems (NMS) are software platforms which provide the functionality and tools to centrally manage communication networks. An NMS platform may be from a small scale Windows based application to a large scale, Unix based distributed application. A new equipment added to the network needs a management application that hooks up to the existing NMS platform currently managing the network. This management application is usually supplied by the network equipment vendor and is used to monitor and control the equipment.

SNMP, Management Information and MIB Files

An important part in the design of a new networking device is the design of the management information that will be available for this new device. SNMP (Simple Network Management Protocol) which is the most prevalant management protocol, defines a syntax for this management information called MIB (Management Information Base). The MIB includes the variables and tables that can be read from (or written to) this device. It also defines other important attributes of this managed information. SNMP Supports GET, SET, GET_NEXT, RESPONSE and TRAP messages to and from the agent. [1],[3]

Following is an example of the MIB file syntax:

```
sysName OBJECT-TYPE
SYNTAX  DisplayString (SIZE (0..255))
ACCESS  read-write
STATUS  mandatory
DESCRIPTION

        "An administratively-assigned name for
        this managed node.  By convention, this is
        the node's fully-qualified domain name."

::= { system 5 }
```

MIB File syntax example

NMS Applications Development Cycle

A networking equipment, includes a software agent which communicates with the management application managing this equipment. In order to develop a management application, one needs the device's MIB definition and a working agent. Since a networking device cannot be shipped without a management application, starting the application development only after the agent is already functional, causes a substantial delay in the availability of the equipment with its management application.

Using a Simulator instead of the Real Agent

Using an agent simulator can help to reduce this delay, by turning the development process of the agent and the management software to concurrent processes, and reducing the overall development cycle. Using a simulation can also improve the final quality of the agent by causing the design problems in the MIB to arise at an earlier stage, when its easier to make changes in the agent software. The simulator can also be used for testing the management software since it is much easier to configure a simulation then to build a real working language for testing. Developing an agent prototype in a high level scripting environment instead of the regular embedded software environment, helps to get better results from the agent in a shorter time.

GeNMSim Main Features

The main features supported by GeNMSim are:

- Automatic creation of a working database from MIB files
- Multiple agents in one GeNMSim process
- User customisation with Tcl callbacks
- Online traffic statistics with GeNMSim 'Probe'
- Multi Platform - runs on Unix and Windows95/NT

Why Tcl ?

Portability

GeNMSim is designed as a portable product for Unix and MS/Windows platforms. Using Tcl as the engine behind GeNMSim avoids many of the portability issues that are encountered in such a product. The main part in GeNMSim for which Tcl does not provide portability is the network interface which is implemented in the Unix version using the SNMP library provided by Carnegie Mellon University and in the Windows version using an agent enabled WinSNMP package. [5]

User Customisation

GeNMSim provides many hooks for user customisation of the agent. Using Tcl as the scripting language for user customisation gives the user all the power of Tcl/Tk without the need to compile or link the program. User scripts written for one operating systems are ported automatically. The user gets all the GeNMSim added Tcl commands of which some are implemented as C functions and others as Tcl procedures.

Ease of Development

Developing a project using Tcl saves substantial R&D time. Making changes to the code does not require compilation and thus much of the programmers idle time is reduced. The user interface part is very simple to build and does not require learning a Motif or an MS/Windows GUI package. The GeNMSim database is also implemented as a set of Tcl scripts which avoids the need to write and maintain a separate parser and allows using the functionality of the Tcl script loading mechanism while loading the data base.

Alternatives

The most important requirement from GeNMSim is its ability to be customised by the user. This requirement can be achieved either by using a script language (like Tcl) or by having the user work in a Compile/Link programming environment. Other scripting languages (such as Visual Basic or Perl) are not portable between Unix and Windows and are hard to customise. Using a C/C++ programming model, forces the user to have a full development environment and also takes us back to the portability issue. [2]

GeNMSim Tcl Based Data Base

General

An SNMP agent simulation requires 3 types of information:

1. The SNMP MIB files which are designed by the networking equipment vendor as part of the overall design, and are shared by both the agent and the manager.
2. Current agent MIB values - These values represent the current state of the simulated agent with current values of the data held in the agent.
3. Agent behaviour - A real agent is characterised by both the information it can provide and the actions it can take when something happens. These actions have to be defined when creating a simulated agent.

Defining a new agent for simulation is done in 2 phases:

1. Define the ***Agent Type***. This can be viewed as defining a new class in object oriented programming. An ***Agent Type*** defines the information and behaviour of a certain agent type.
2. Define the ***Agent Instance***. This can be viewed as defining an object of a given class in object oriented programming. The ***Agent Instance*** includes the current state and values of a specific agent being simulated.

Agent Type Data Base

The process of creating a new ***Agent Type*** is divided into two parts:

1. Automatically extract the information contained in the MIB files which define the management scope of the simulated agent. This process is carried out via the ***Create Agent*** tool which is described later.
2. Manually configure the new agent type by adding the agent behaviour via callback functions. Callback functions are Tcl procedures that are called at certain points of the simulation. Callbacks are described in further detail later on.

Lets assume we're defining an agent type named **MyAgentType.** The ***Agent Type*** is defined by 4 ASCII files which are actually Tcl scripts.

The first and most important file is **MyAgentType.def** file which includes the definition of MIB information as extracted from the MIB files.

Figure 1 shows an example of the .def file:

The keyword **Sim_Db_loadMibDef** is actually a Tcl procedure which adds an attribute for the specified MIB object to the data base.

The second file is **MyAgentType.oid** which defines translation from ascii names to ASN.1 decimal notation which is required for the simulation process. This file is also a Tcl script in which each line is a call to a procedure to add a <name,oid> pair in the data base.

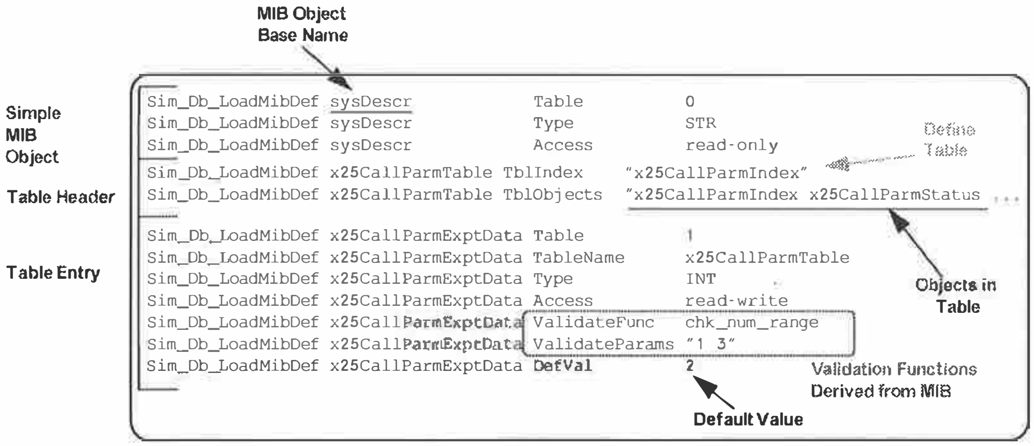
An important part of the agent behaviour is to send SNMP traps which are asynchronous messages sent to the manager with accordance to the agent behaviour. The third file named **MyAgentType.traps** defines the traps supported by this agent and are also automatically extracted from the MIB definition files.

The fourth and last file is **MyAgentType.user_def** . This file contains all the callback registration commands for this agent type. This file has the same syntax as .def files. The reason for seperating the user additions from the automaticlly created file is in order to allow easier migration in subsquent creation of the AgentType database and in new GeNMSim versions, since this file is not erased when the database is recreated.

Agent Instance Data Base

The ***Agent Instance*** is also built in two phases:

1. As part of the automatic process, a template of the ***Agent Instance*** is created to allow quick start for the agent simulator user.



```
Sim_Db_LoadMibDef sysDescr                Table          0
Sim_Db_LoadMibDef sysDescr                Type           STR
Sim_Db_LoadMibDef sysDescr                Access         read-only
Sim_Db_LoadMibDef x25CallParmTable TblIndex     "x25CallParmIndex"
Sim_Db_LoadMibDef x25CallParmTable TblObjects   "x25CallParmIndex x25CallParmStatus ...
Sim_Db_LoadMibDef x25CallParmExptData Table          1
Sim_Db_LoadMibDef x25CallParmExptData TableName      x25CallParmTable
Sim_Db_LoadMibDef x25CallParmExptData Type           INT
Sim_Db_LoadMibDef x25CallParmExptData Access         read-write
Sim_Db_LoadMibDef x25CallParmExptData ValidateFunc   chk_num_range
Sim_Db_LoadMibDef x25CallParmExptData ValidateParams "1 3"
Sim_Db_LoadMibDef x25CallParmExptData DefVal         2
```

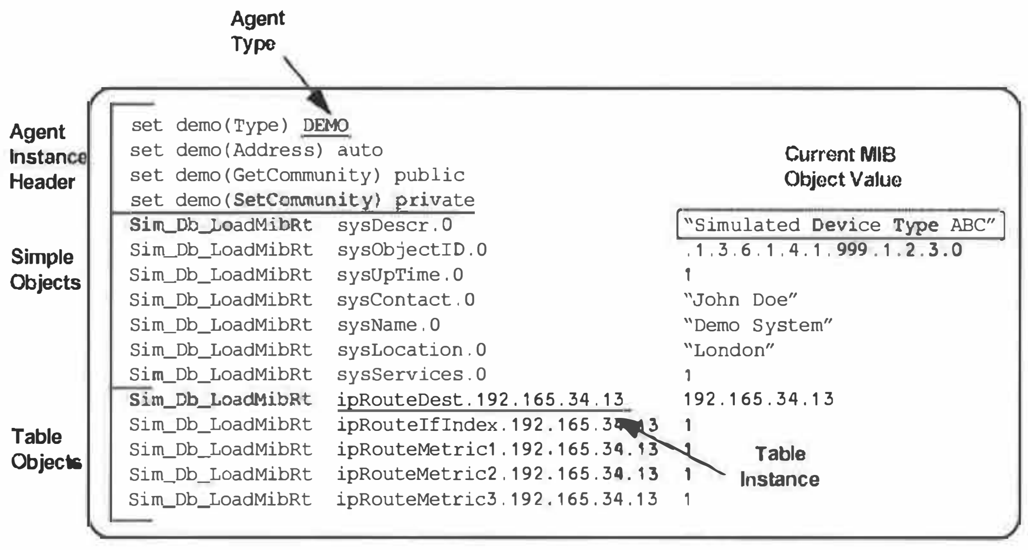
Figure 1: AgentType .def file example

2. This template is modified by the user (currently via a text editor) to reflect the real (initial) values of the simulated agent and are updated by the GeNMSim runtime as values in the database are modified.

Lets assume we're defining an agent instance named **MyAgent.** This instance is of type **MyAgentType.** The *Agent Instance* is defined by 2 more Tcl scripts.

The first and most important file is **MyAgent.rt** file which includes the current values of the database tables.

Figure 2 is an example of the .rt file.



```
set demo(Type) DEMO
set demo(Address) auto
set demo(GetCommunity) public
set demo(SetCommunity) private
Sim_Db_LoadMibRt  sysDescr.0                    "Simulated Device Type ABC"
Sim_Db_LoadMibRt  sysObjectID.0                 .1.3.6.1.4.1.999.1.2.3.0
Sim_Db_LoadMibRt  sysUpTime.0                   1
Sim_Db_LoadMibRt  sysContact.0                  "John Doe"
Sim_Db_LoadMibRt  sysName.0                     "Demo System"
Sim_Db_LoadMibRt  sysLocation.0                 "London"
Sim_Db_LoadMibRt  sysServices.0                 1
Sim_Db_LoadMibRt  ipRouteDest.192.165.34.13     192.165.34.13
Sim_Db_LoadMibRt  ipRouteIfIndex.192.165.34.13  1
Sim_Db_LoadMibRt  ipRouteMetric1.192.165.34.13  1
Sim_Db_LoadMibRt  ipRouteMetric2.192.165.34.13  1
Sim_Db_LoadMibRt  ipRouteMetric3.192.165.34.13  1
```

Figure 2: Agent Instance .rt file example

The keyword **Sim_Db_loadMibRt** is actually a Tcl procedure which adds an entry for the given MIB object to the data base.

The second file is **MyAgent.managers** which defines the addresses of the managers which will receive traps generated by this agent.

GeNMSim Offline Tools

There are several offline tools in GeNMSim which help in preparing the simulation. We'll concentrate on the **CreateAgent** tool which plays an important role in GeNMSim and is the most interesting in terms of Tcl. This tool is a Tcl program which parses MIB files and generates the automatic parts of the GeNMSim database.

Parsing the MIB Files

A MIB file is a text file in a standard format defining **Attributes** (Types, Access, Syntax etc.). MIB files may import definitions from other MIB files (such as include files). Each MIB object defined in the MIB file has a unique location in the global **MIB tree.**

The following is an example of the MIB file syntax:

```
iso      OBJECT IDENTIFIER ::= { 1 }
org      OBJECT IDENTIFIER ::= { iso 3 }
dod      OBJECT IDENTIFIER ::= { org 6 }

ObjectSyntax   ::= INTEGER
ObjectName     ::= INTEGER
internet    OBJECT IDENTIFIER ::= { iso org(3)
                                    dod(6) 1 }
directory    OBJECT IDENTIFIER ::={ internet 1 }
mgmt         OBJECT IDENTIFIER ::={ internet 2 }
experimental OBJECT IDENTIFIER ::={ internet 3 }
private      OBJECT IDENTIFIER ::={ internet 4 }
enterprises  OBJECT IDENTIFIER ::={ private 1 }

sysLocation OBJECT-TYPE
SYNTAX  DisplayString (SIZE (0..255))
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"The physical location of this node (e.g.,
`telephone closet, 3rd floor')."
::= { system 6 }
```

The ***CreateAgent*** program parses this text, extracting from it the MIB **Name**, **Syntax**, **Access** and its **Location** in the MIB tree. If the MIB object is part of a **Table**, the table information is also resolved. In order to parse these files, the Tcl **string** and **regexp** commands are used extensively. In the example above, we can see the SYNTAX of the MIB object is '`DisplayString (SIZE (0..255))`' meaning it is a string with maximal length of 255 characters. In this case, a validation callback function will automatically be registered to check that the values set to this object do not exceed the limits in the definition. The same scenario applies to other conditions that may appear for MIB object values.

The ***CreateAgent*** is actually comprised of several Tcl scripts each responsible for one part of the agent database. These scripts are called as separate tasks using the Tcl exec command in order to allow an independent global variable scope for each of the scripts. See also the sec tion on portability later on.

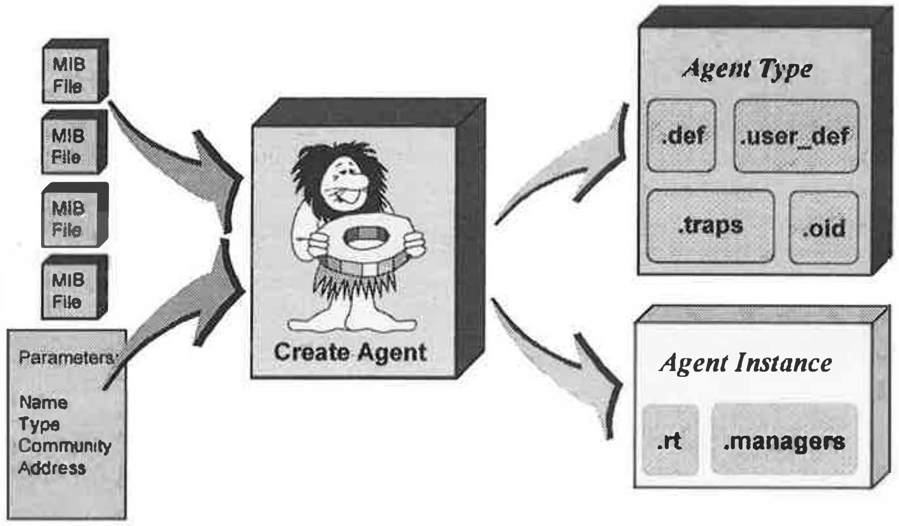


Figure 3: Create Agent - Input and Output files

Figure 3 shows the input and output of the ***CreateAgent*** program.

The GeNMS Technology

Overview

Since GeNMSim is based on the GeNMS technology, it is important at this point to take a brief glance on the GeNMS technology. GeNMS is a Tcl/Tk based technology which provides a framework for creating network management products, applications and tools. GeNMS includes many mechanisms intended to allow easy development of these products. Tcl/Tk gives it the GUI portability and allows for most of the OS independence. GeNMS also supports portability over NMS platforms and management protocols.

Message & Trap handling

Messages, Which are one of the GeNMS mechanisms, are implemented as a C structure and function library which has a Tcl interface. A message is represented in Tcl as a string handle. There are API functions to create new messages, add or fetch variables, read or update header parameters and forward the message to the network. A message received from the manager via the network interface, is passed to a message dispatching mechanism which then registers the message (in a Tcl hash table) and forwards the message to the simulator handling function (in the case of GeNMSim) which is implemented fully in Tcl. The simulator processes this message by extracting the request code and variables and accessing the data base as requested. The response message is then built and send via the GeNMS scheduling mechanism (Tcl TimerHandler based scheduling mechanism) back to the manager via the network and platform layers.

GeNMSim Runtime

The GeNMSim runtime is a GeNMS/Tcl/Tk based executable enriched with numerous application specific commands.

The process starts with an initialisation phase which loads the data base files, starts the network interface, creates the GeNMSim windows and prepares for message processing. This includes creating a new socket and file handler and adding this file handler via the Tcl FileHandler mechanism to the Tcl select loop. Since SNMP is based on UDP, there's usually a well known UDP port to which an SNMP agent listens to.

When initialisation is complete, GeNMSim is ready to receive SNMP and to 'act' as a real agent.

GeNMSim Traps

Traps are generated in GeNMSim callback functions or as a result of a delayed action which is activated by the scheduling mechanism. The trap is a message which originates at the simulator level and is sent to the network.

Using Tcl Data Structures for holding data

Most of the agent's internal data is held in Tcl two dimensional arrays. Since GeNMSim can support more the one concurrent agent, one of the array dimensions is the agent type or agent name and the other index is the attribute. Each MIB name has its own attribute array and there are several general purpose arrays to hold the name to oid translation and a linked list of the objects currently held in the agent.

Callback Type	**Description**
PreInit	Called when GeNMSim is started before loading the data base
PostInit	Called after data base is loaded
PreGet	Called upon a get request before the GET is done
Get	Called upon a get request instead of the standard GET processing
PostGet	Called upon a get request after the GET is done
PreSet	Called upon a set request before the SET is done
Set	Called upon a set request instead of the standard SET processing
PostSet	Called upon a set request after the SET is done
Validate	Called upon a set request to check if the value to be set meets field validation criteria
ValidateGroup	Called upon a set request after each one of the object validation callbacks are processed to validate mutual dependencies

Figure 4: GeNMSim Callback Types

Customising GeNMSim with Tcl based Callback Functions

What is a Callback Function

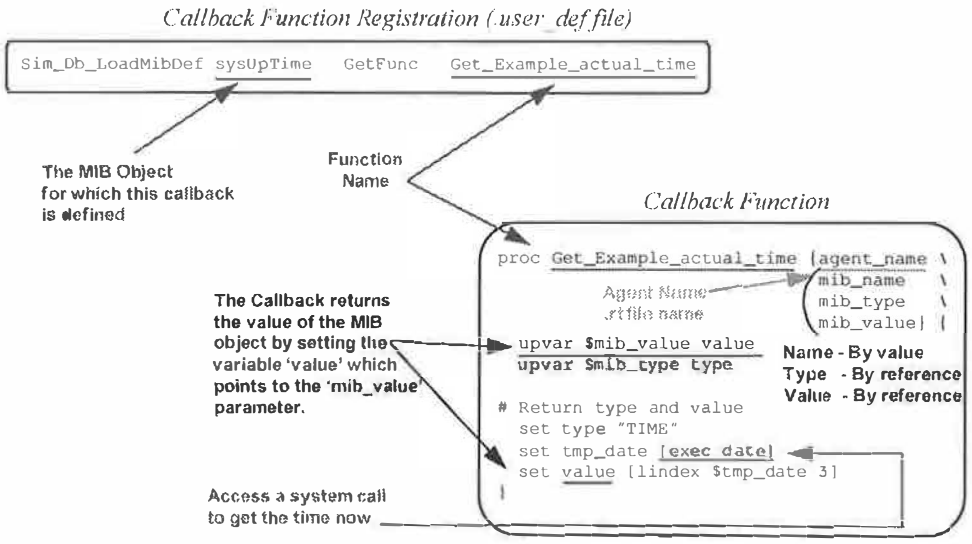
Customising an application can be done is several methods. One of the most common ways is to give the application user, the hooks to add procedures that will determine the behaviour of the application at that point. In order to add a new callback to GeNMSim, the user has to write this callback in a designated directory (in Tcl) and then register the callback in the database and associate it with the desired MIB object and callback type.

Types of Callback Functions

GeNMSim supports 10 types of Callbacks. Figure 4 lists the supported callback types.

Callback Registration and Usage Example

The following illustration shows an example of a callback function. This example is of a Get callback for the MIB object ***sysUpTime*** which generally means 'how much time has this systems been running'. In our case, the callback function returns the current time from the system. 1st The callback is registered in the **.user_def** file. The callback itself uses the Unix '**date**' command to get the current time. Another way to implement this callback in a more portable way is to use the Tcl '**clock**' command.


Callback Function Registration (.user_def file)
The MIB Object for which this callback is defined
Function Name
Callback Function
Agent Name .rtfile name
The Callback returns the value of the MIB object by setting the variable 'value' which points to the 'mib_value' parameter.
Name - By value
Type - By reference
Value - By reference
Access a system call to get the time now

```
Sim_Db_LoadMibDef sysUpTime    GetFunc    Get_Example_actual_time
```

```
proc Get_Example_actual_time {agent_name \
                              mib_name   \
                              mib_type   \
                              mib_value} {
  upvar $mib_value value
  upvar $mib_type type

# Return type and value
  set type "TIME"
  set tmp_date [exec date]
  set value [lindex $tmp_date 3]
}
```

Figure 5: Callback registration and implementation

Using GeNMSim Database API from Callbacks

It often occurs that callbacks need to have access to the GeNMSim database. In such a case, the callback has access via the database API functions which is a set of Tcl procedures or commands. An example of a callback function that makes use of some of these API functions is listed below. In this example, the **PostGet** callback increments a counter in the MIB. The name of the counter is passed to the callback by the calling mechanism. This information is available from the callback registration.

```
proc Ex_PostGet_Increment_Counter {agent_name \
              mib_name mib_type \
              mib_value args} {
# Extract counter name from the 'args' variable
  set cnt_name [lindex $args 0]

# Get the current value of the counter.
  if {[Sim_Db_GetMibVal $agent_name $cnt_name \
      cnt_value] != 0} {
    puts "--> Error while getting value of  \
        object $cnt_name"
    return
  }

# Get the type of the incremented object
  Sim_Db_GetMibType $agent_name $cnt_name \
          cnt_type

# Set the new value (old value + 1) to cnt_name
  Sim_Db_SetMibVal $agent_name $cnt_name \
      $cnt_type [incr cnt_value]

# Save the run-time values to the rt file
  Sim_Db_SaveRtValues $agent_name
}
```

A Callback using GeNMSim database API functions

GeNMSim Windows

GeNMSim has several windows implemented in Tk to display statistics of the traffic which passes through the agent. The windows include a general statistics window, a message list window and a message details window. Since these are pretty simple Tk windows, There is no need to discuss them in detail.

Conclusions and Future Directions

Performance

An SNMP agent can in some cases be a computation intensive application. These cases may be when a GET_NEXT command is applied for a MIB object which is currently not present in the database. (GET_NEXT means, fetch the object which follows the given object in the MIB tree). Since implementing a tree data structure in Tcl is far from optimum, GeNMSim holds the objects in Tcl arrays with linked lists giving the order of the variables. In a test case, made on a simulated agent with about 200 entries, a GeNMSim agent responds after 3 seconds, while a real agent responds in only a fraction of a second. In a larger agent the performance penalty is linear since the search is performed sequetially. If we'll consider a table with 5 columns (a typical case) which is located in an agent of 500 lines, then a GET_NEXT to this table when it is empty, using GeNMSim may take up to 25 seconds (5 columns X 5 seconds each). This kind of a response from an agent cannot be considered acceptable. In most cases though, fetching or updating a variable from the database is done in reasonable time. To improve performance, we're now considering to add a C implemented tree structure for the MIB tree.

Portability

GeNMSim was initially developed for SunOS with Tcl 7.5 and Tk 4.1 . Most of the GeNMSim Tcl code has been ported with no changes to Windows95/NT with several exceptions:

1. **Unix file access systems calls** - Commands like **chmod** and **cat**, which have no Tcl commands to 'cover' them, were used in the Unix version but could not be ported to the windows version.
2. **Unix environment variables** - The **env** array which contains the system environment variables was extensively used in the original Unix version. This feature exists on the Windows/NT system but in Windows/95, which is based on DOS, environment variable names are case insensitive and there is a problem with allocation of environment space for the environment variables. To overcome these problems, most of the environment variables where switched with configuration variables in a Tcl global array.
3. **Tcl exec command** - The **exec** command was used in the Unix version in the CreateAgent tool to invoke the sub-tools in separate shells. Since the Windows version does not support exec (Tcl version 7.5), we used **multiple interpreters** in which we started the sub-tools. Slave interpreters, in combination with the **alias** command, allowed invocation of sub-tools in their own scope of global variables.

Other then the Tcl code portability issues, there was a need to rewrite the networking layer to support WinSNMP. Since GeNMS is a modular technology, this task was simple (about weeks of programming).

Development Tools

GeNMSim is a pretty large project. It contains over 10,000 lines of Tcl code and a large number of C code lines. A good, source level debugger for Tcl (plain vanilla Tcl/Tk) would have been much help during the process of debugging GeNMSim.

Performance analysis is also an important issue when creating a computation intensive Tcl program.

GeNMSim makes use of the debugging aids that are part of the GeNMS technology. These include a sophisticated debug printings mechanism, a Tk widget configuration dump and the GeNMS Classes & Objects mechanism which is beyond the scope of this paper.

The Future of GeNMSim

GeNMSim is one of a family of products that Milestone is creating for the NMS market. The next version of GeNMSim will include an offline graphic configuration tool to edit and configure the database. There is also a thought about supporting other management protocols besides SNMP.

Availability

GeNMSim is a commecial product. For more information contact (via email): GeNMS@milestone.co.il.

Evaluation copies available upon request.

Refrences

[1] Marshall T. Rose. *The Simple Book: An Introduction to Internet Management* (2nd edition). Prentice Hall, 1993. ISBN 0-13-177254-6

[2] Marshall T. Rose, Keith McCloghrie. *How to Manage Your Network Using SNMP*. Prentice Hall, 1995. ISBN 0-13-141517-4

[3] William Stallings. *SNMP, SNMPv2, and CMIP. Practical Guide to Network-Management Standards.* Addison-Wesley, 1993, ISBN 0-201-63331-0

[4] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994/ ISBN 0-201-63337-X

[5] Eric F. Johnson. *Graphical Applications with Tcl & Tk.* M&T Books, 1996. ISBN 1-55851-471-6

The Tycho User Interface System

Christopher Hylands, Edward A. Lee, H. John Reekie
School of Electrical Engineering and Computer Sciences
University of California – Berkeley
Berkeley CA 94720
{cxh, eal,johnr}@eecs.berkeley.edu

Abstract

Tycho is the next-generation user-interface system we are building for the Ptolemy project. It is a complete [incr Tcl] application structured as an extensible class library. Our goal is to make it easy to extend this basic application with functionality and a user interface for specialized applications such as electronic design and simulation. The Tycho library includes a selection of general-purpose widgets, syntax-sensitive text editors, and graphical editing support. It incorporates architectural features that make it easy for different editors and viewers to share data and screen space. Finally, structured support for incorporating C and Java packages into this framework allows us to use those languages to complement the scripting and user-interface features of Tcl/Tk.

1 Introduction

Tycho [7] has grown from our frustration with the user-interface facilities of the present version of Ptolemy, a large C++ software package that is used to design, simulate, and generate signal processing and communications systems [6, 8, 1]. In the summer of 1995, we began to explore [incr Tcl]/[incr Tk] [11] (then at version 1.5) as a potential candidate for replacement of the Ptolemy user interface. As the project evolved, the focus of the project has shifted away from merely a replacement user interface, towards providing a framework within which we can integrate Ptolemy, new user-interface features, new simulation sub-systems coded in Java, and documentation for all of these.

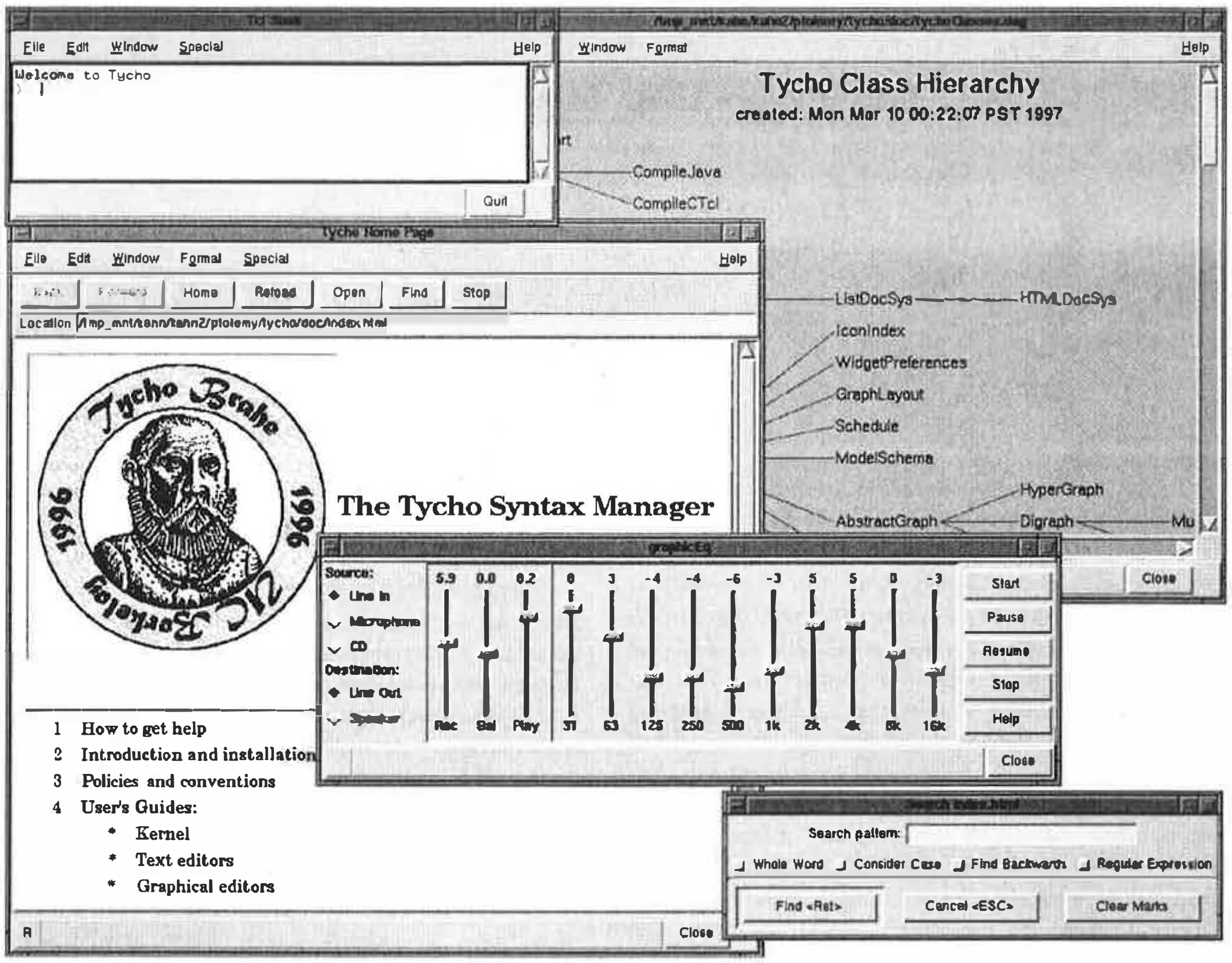
Ptolemy, written mostly in C++, runs on a dozen flavors of Unix. It currently contains over 350,000 lines of code, and maintaining and building releases that support all of these platforms in a system this size is very resource-intensive (especially for an academic research group). And we still can't support Windows or Macintosh. [incr Tcl]/[incr Tk] (and Java), on the other hand, seem to provide an ideal opportunity to get out of the "binary-building business" and into the "platform-independent software business."

The [incr Tcl]/[incr Tk] part of Tycho is structured as a reusable and extensible framework of classes. Tycho will run in a vanilla **`itkwish`**, although we also have binaries for Tycho-with-Java, and Tycho-with-Ptolemy. A small number of C-code packages provide support we need in specific applications, such as access to real-time timers. Tycho has been developed on UNIX platforms and most of it works under Windows NT; the Macintosh port is functioning but unreliable.

Tycho is a complete application, not just a library of widgets. By default, Tycho starts up with a welcome window and its own Tcl shell, with menus that give access to all of its functionality (figure 1). It is, however, an *extensible* application: wherever possible, we have tried to build class hierarchies that allow developers to inherit or compose key functionality, so that they need to provide only the additional functionality needed for their application. We use Tycho as our [incr Tcl] development environment, and anticipate that on-going development of C and Java compilation support will make Tycho key infrastructure for all research in the Ptolemy group by the end of this year.

2 Background and motivation

Ptolemy is a software package that is used to design signal processing and communications systems, ranging from designing and simulating algorithms to synthesizing hardware and software, parallelizing algorithms, and prototyping real-time systems. Ptolemy was started in 1990, with version 0.7 scheduled for release in June 1997. Ptolemy has hundreds of active users in industry,


File Edit Window Special Help
Quit
Window Format Help
CompileJava
CompileCTcl
ListDocSys
HTMLDocSys
IconIndex
WidgetPreferences
GraphLayout
Schedule
ModelSchema
HyperGraph
AbstractGraph
Digraph
Close
File Edit Window Format Special Help
Home Reload Open Find Stop
Tycho Brahe 1996 1996 ZIC Berkeley
Source: Line In Microphone CD
Destination: Line Out Speaker
Rec Bal Play 31 63 125 250 500 1k 2k 4k 8k 16k
Start Pause Resume Stop Help Close
Search pattern:
Whole Word Consider Case Find Backwards Regular Expression
Find <Ret> Cancel <ESC> Clear Marks

Welcome to Tycho

Tycho Class Hierarchy
created: Mon Mar 10 00:22:07 PST 1997

Location /tmp_mnt/tenn/tenn2/ptolemy/tycho/doc/index.html

The Tycho Syntax Manager

1 How to get help
2 Introduction and installation
3 Policies and conventions
4 User's Guides:
  - Kernel
  - Text editors
  - Graphical editors

Figure 1: Tycho in action

academia, and government, and an active newsgroup (`comp.soft-sys.ptolemy`).

Ptolemy's current graphical editor is based on the VEM package developed at Berkeley in the mid-80's. Figure 2 shows the block-diagram editor. Users construct simulations by placing and connecting icons representing signal processing components on the screen. Because of its age, this interface is clumsy, hard-to-use, and out of step with modern user expectations. One of Tycho's original design goals is to replace this outdated interface, and Tcl/Tk was a clear choice for us.

Tcl/Tk was added to Ptolemy in 1992 to provide interactive and animated simulation runs. Since late 1996, Ptolemy has been running the [incr Tcl] interpreter instead of Tcl, and all new user interface work in Ptolemy is now done in the Tycho framework. (Note, however, that Tycho can also be run independently of Ptolemy.) Currently, Tycho and the older graphical editor are used together. While the older editor is still used for the block diagrams, Tycho's text editors and HTML viewer are used to browse and edit block definitions and documentation. Tycho's Tcl Shell can be opened from within Ptolemy, which provides a command-based Tcl interface to the Ptolemy kernel. We hope to complete the transition to the Tycho user interface over the summer.

One of the key factors in the success of the Ptolemy project has been its non-dogmatic support of multiple semantic models. That is, it encourages designers to produce a *domain* that encapsulates a specific model of computation and to interface it with other domains. Thus, Ptolemy supports various dataflow models, discrete-event simulation, finite-state machine models, and others. Tycho aims to provide a similar level of support for the syntactic component of system design: a designer should be able to rapidly construct a user interface to represent a particular semantic model or to vi-

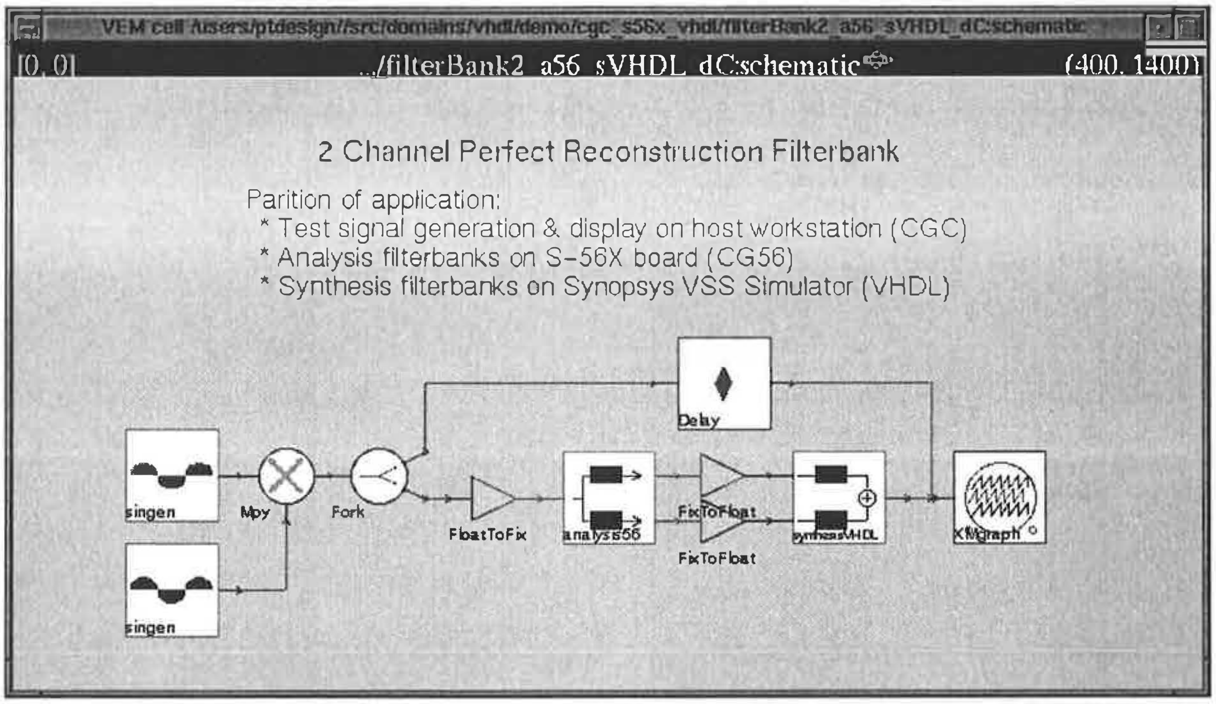

VEM cell /users/ptdesign//src/domains/vhdl/demo/cgc_s56x_vhdl/filterBank2_a56_sVHDL_dC:schematic
[0, 0]
../filterBank2_a56_sVHDL_dC:schematic
(400, 1400)
2 Channel Perfect Reconstruction Filterbank
Parition of application:
* Test signal generation & display on host workstation (CGC)
* Analysis filterbanks on S-56X board (CG56)
* Synthesis filterbanks on Synopsys VSS Simulator (VHDL)
Delay
singen
Mpy
Fork
FloatToFix
analysis56
FixToFloat
FixToFloat
synthesisVHDL
XMgraph
singen

Figure 2: Ptolemy's current block diagram editor

sualize application-specific design information. For researchers in our own group, we aim to provide the support needed to rapidly construct customized user interfaces to signal processing and communications simulations. More broadly, Tycho has become an expression of our ideas about Tcl/Tk development environments: we want to provide an extensible framework in which mundane tasks such as documentation generation and indexing, font management, color management, and dialogs with the user are built using a shared, common infrastructure.

3 Tycho's development support

One of Tycho's goals is to make it easy to use Tycho itself as our development environment. To this end, we have incorporated interfaces to a few of the "most useful" development tools into Tycho. All of Tycho and Ptolemy is indexed by Glimpse [10], and this can be brought up from any Tycho window. We use SCCS revision control on all our sources, and so have a revision control dialog window that provides the most commonly-used commands, as well as the ability to regenerate any past history on demand. (Although we use SCCS, Tycho supports RCS equally well.) An off-line script processes every file in the documentation tree to produce various indexes of the documentation – every Tycho window can bring up viewers into these indexes to jump to needed documentation. A class browser (shown in figure 1) provides on overview of and access to every class in the system. Any text editor can bring up a spell-checking interface.

Tycho includes a number of syntax-sensitive text editors. We are not trying to duplicate the functionality of, say, **emacs**, but to provide a solid set of basic features which can work in concert with graphical interfaces and the integrated documentation system. Language-specific classes extend the basic editing facilities with support for compilation (C, C++ and Java), automatic documentation generation (Tcl and [incr Tcl]), and language-specific file, class, and function templates. Again, this is made possible by inheritance. Often, customizing functionality is simply a matter of inheriting from a suitable class and overriding one or two methods. For example, we have a shell that "looks inside" and monitors all activity with a *model* object (see section 4.1). This class inherits from the *TclShell* class and overrides just one method: *evalCommand*, which processes the text of an input command.

A great deal of Tycho's documentation is extracted from its source code. We use a similar scheme to Sun's Javadoc system [4], which extracts documentation from comments preceding class and method declarations, and uses special tags to distinguish different fields of the comment text. Text within comments is formatted in HTML, as is the generated documentation and indexes. All documentation can be viewed from within Tycho or using an external browser such as Netscape. In Tycho, anything can have a hyperlink to anything else – documentation to sources and vice versa; graphical editors to textual; and so on.

The Tycho HTML widget, based on Sun's HTML parsing library and shown in figure 1, has some enhancements that may be useful to other Tcl developers. First, the library relies on calls to `unknown` to handle unrecognized HTML tags. By defining all possible tags with null operations where necessary, HTML parsing sped up by a factor of three! Another performance problem was related to scoping in mega-widgets: we needed to create the text widget component at the global scope in order to prevent calls to `unknown`. Second, the HTML widget supports a `tcl` tag, which marks executable Tcl – any text so marked can be executed by double-clicking on it. This simple extension is surprisingly useful. For example, every class file contains a few lines of executable Tcl to demonstrate how the class may be used, which serves as both an example to a documentation reader and as a confidence check that documentation and code are up to date. We have taken this further where appropriate, and have written on-line tutorials for most of the user-interface support classes, which can be executed by reading and clicking.

4 Architectural patterns

4.1 The Model-View pattern

Model-view is a derivation of Smalltalk's model-view-controller (MVC) architecture; the model-view derivation combines MVC's view and controller into a single abstraction [2]. In Tycho, we have implemented a *Model* class that provides a publish-and-subscribe mechanism, unbounded history, a simple but flexible external structured file format called TIM (Tycho Information Models), and a simple serialization mechanism. Its subclasses implement application-specific models, such as storing user preferences, information about classes used by the documentation generator and class browser, indexes into the file system, and so on. There are also a set of graph classes which, although currently written in Itcl, we hope to eventually have written in Java.

Models are based on TIM (Tycho Information Models), which is a meta-data format that is intended to encourage clean representations of data, both in in-memory objects and in an external file representation. It is loosely based on the concepts of Object Modeling Technique (OMT) [14]: a model is a collection of entities and links between entities. Each entity and link has a unique name, value, and a list of attributes. Entities can be nested, so TIM is naturally hierarchical. A small TIM (for a dataflow graph) is:

```
vertex a {
    port out -tokencount 2 -type output
}
vertex b {
    port in-0
    port in-1
    port out -type output
}
edge a out b in-0 -initialdelay 0
```

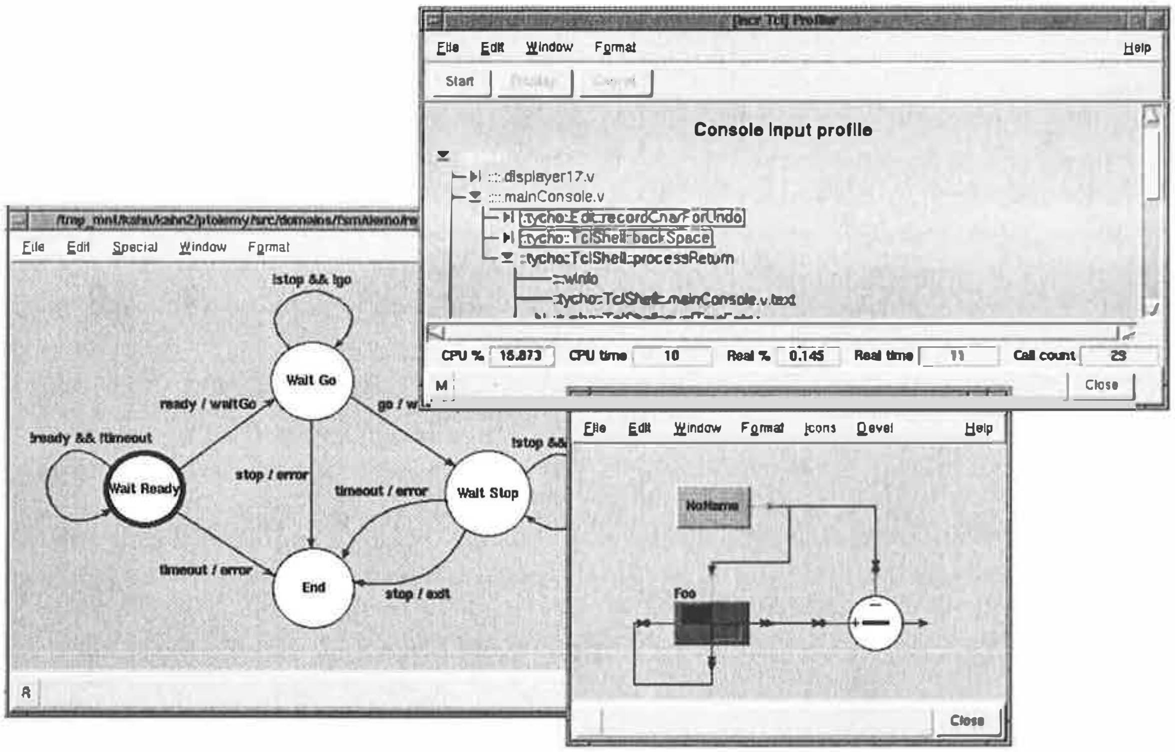
Models contain a straight-forward implementation of the publish-and-subscribe pattern (also known as the *Observer* pattern [5]): any view that is interested in a model can subscribe to it, and will be notified of any updates to the model. The prototype graph editor in figure 3, for example, has two models: one for the graph, and one for the canvas layout. We have demonstrated multiple views editing the same two models. The concept is pervasive throughout Tycho – all major widgets, for example, are subscribed to the user preferences model. When a user changes, say, the font used in menu buttons, all menus are automatically updated with the new font.

Because user interfaces typically allow a user to undo and redo operations, Model implements a reasonably flexible, unbounded, undo and redo mechanism. Each method that changes a model is required to return a script that will undo the change. Both the method call and arguments and the undo script are stored in linear arrays.

4.2 The Displayer-View pattern

Early in its life, Tycho had two classes for every type of editor: one for the top-level window with menu and key bindings, and one containing the editing widget. We simplified this considerably by adopting a pattern we call Displayer-View, in which there is only one top-level window class (the Displayer). One or more views can place themselves into the Displayer and request access to a menu bar, tool bar, and status bar. The HTML window in figure 1 shows all three of these bars.

This approach gives us the flexibility to create widgets that can be placed into Displayers in new



[incr Tcl] Profiler
File Edit Window Format Help
Start
Console input profile
::displayer17.v
::mainConsole.v
::tycho::Edit::recordCharForUndo
::tycho::TclShell::backSpace
::tycho::TclShell::processReturn
::winfo
::tycho::TclShell::mainConsole.v.text
CPU % 18.873 CPU time 10 Real % 0.145 Real time 11 Call count 23
Close
File Edit Special Window Format
!stop && !go
Wait Go
ready / waitGo
go / w
!ready && !timeout
Wait Ready
stop / error
timeout / error
Wait Stop
!stop &&
timeout / error
End
stop / exit
File Edit Window Format Icons Devel Help
NoName
Foo
Close

Figure 3: Some of Tycho's graphical editors

combinations. For example, a member of our group (Cliff Cordeiro) is working on a class browser that combines an HTML with a class index. We anticipate many more uses of this pattern in the future.

5 Graphical editing support

Although Tycho was originally intended only to serve as a user interface to Ptolemy. We now see Tycho as an opportunity to experiment with design visualization, broadening the perspective beyond a block-diagram perspective, and exploring new visual and mixed visual/textual syntaxes for design representation and understanding.

The infrastructure to support this vision falls into two categories. First is a set of classes that implement sophisticated canvas-like capabilities: hierarchical canvas items, abstractions for user interactions, and a graphical selection analogous to the textual selection of the Tk text widget. The second is a set of complete graphical editors. The editors are still early in development, but already include a simple finite-state-machine editor, a graphical class hierarchy display, and a graphical interface to the Tcl profiler from the TclX package [9] (see figures 1 and 3).

As much as possible, we have tried to capture functionality needed by typical graphical editors in Itcl classes. Tycho's enhanced canvas, which we call a *slate*, adds hierarchical items to Tk's standard canvas. The slate is implemented entirely in Itcl, and so should work with other canvas extensions such as the Dash patch [13]. The slate uses canvas tags to enable a collection of canvas items to be treated as a single complex item. All canvas commands functions correctly with single canvas items or complex slate items. Each type of complex item requires that a fairly small class be written to implement operations such as `create` and `coords`. We have a small library of such items, such as 3D rectangles and polygons, items with labels and scalable graphics on them (which we use for icons in graphical block diagrams), and self-routing lines (again for block diagrams).

Figure 4 illustrates a hierarchy of visual items. The root item contains a number of items within itself, one of which is an "icon" item that recursively includes two other complex items. It also contains a text label and a "terminal" item for connecting lines. The semantics of bindings within this visual hierarchy is not as simple as first appears: if the top-level item has tag *foo* and the terminal has tag *bar*, should the terminal also respond to events bound to *foo*? We have chosen to answer "no." So far,

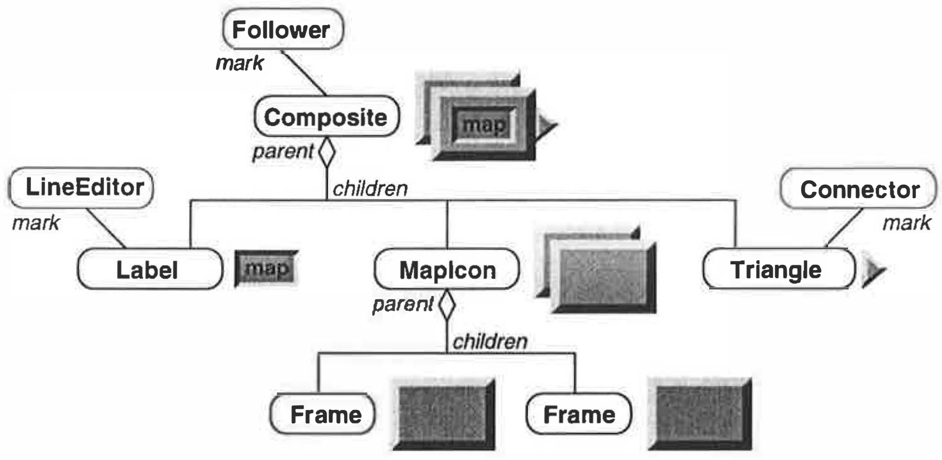


Figure 4: A hierarchical item marked with interactors

our experience indicates that this is the right decision both for implementation and use. In this example, the terminal items will therefore respond to events differently to the other items that make up the icon: if the mouse is dragged over a terminal, a new arrowed line is drawn to be connected to another terminal, whereas dragging the mouse on any other part of the item moves the whole icon, including the terminals and the ends of connected lines.

Figure 4 also shows user interaction objects, called *interactors* [12]. Each interactor captures a particular pattern of interaction. For example, a *Follower* class follows the mouse – by default, attaching an interactor to a canvas tag with a statement like `$follower bind icon` will make all items tagged "icon" draggable with the mouse. Of course, this is not difficult to do with Tk canvas bindings – but by encapsulating this into a class, we can inherit and compose interactors to produce more complex interactions. For example, subclasses of *Follower* keep items within a certain area of the screen, snap movement to a grid, and implement drag-and-drop mechanisms (the *Connector* in the figure). A *LineEditor* interactor edits text labels; a *Selector* interactor implements a graphical selection mechanism similar to that found in Macintosh drawing packages. Combined with the model-view architecture (section 4.1), we believe the visual hierarchy and interactors provide a powerful toolkit for building complex interactive graphical editors.

6 Tycho and Java

We have been exploring the integration of Tycho with Java. The combination of a high-level object-oriented scripting language, (Itcl) with an object-oriented system programming language (Java) is very attractive. Java in itself is attractive for its clean syntax and semantics, its support for distributed programming, and its high degree of portability. Members of the Ptolemy group are currently implementing an exploratory dataflow simulation engine in Java, which will be accessed from a Tycho graphical interface.

We based our Tycho-Java interface on version 0.4 of Sun's experimental Tcl-Java interface [15]. Not all features of Tcl work correctly in this implementation—`exec`, for example, is broken, and the exception handling does not work correctly with JDK1.1 – but it works well enough for us to explore the integration of Tcl and Java. We layered our own Java classes above the Tcl-Java interface to provide a simpler Tcl interface to Java programmers. We register a Tcl command called `::java::new`; this command creates a Java object of the requested class, a wrapper object that converts arguments into the appropriate types, and a Tcl command to access the object. For example, in Tcl, we could write:

```
::java::new tycho.TclExample foo
```

which create the Java objects and registers the Tcl command `foo`. In the TclExample class (in the Java package named `tycho`), a method might look like this:

```
public Double example1(Interp interp,
               Integer iobj, Double dobj ) {
    int i = iobj.intValue();
    double d = dobj.doubleValue();
    return new Double(i + d);
}
```

To call the method, we simply execute the Tcl code:

```
foo example1 5 3.2
```

which prints "8.2" as its result. The wrapper object uses Java's reflection API (`java.lang.reflect`) to find out the argument types of the method being called, converts the strings passed from Tcl into objects of the correct types, and then calls the correct method. Although there is some overhead in this approach, the interface is much cleaner than that provided by the Tcl-Java interface alone. With the Tcl8.0 byte-compiler, we think this style of interface will probably be needed to take advantage of Tcl native types.

As an alternative experiment, we used **tksteal** [3] to re-parent the Java **appletviewer** program into **itkwish**, but found that **appletviewer** didn't have the proper command-line interface to make this work smoothly.

In general, we would like to use [incr Tcl] (and [incr Tk]) for our user interface components, and Java for all new "back-end" development, including complex data structures and simulation tools. Although we have considered the possibility of using Java only, we would much prefer to integrate [incr Tcl] and Java. Apart from saving us a lot of rewriting (Java was still very new when Tycho was started), the Java user interface support is not as mature as Tk, and we rely heavily on Tk's more complex widgets, such as the text widget and the canvas widget.

In our view, it is unfortunate that Sun has not made a bigger effort to integrate Java and Tcl/Tk. Tk is unquestionably superior to Java's current UI support, and Java is the logical choice for future non-UI development. The Java UI is developing rapidly, however, and if we cannot successfully integrate Java and Tcl/Tk, then we may end up moving away from Tcl/Tk as the Java UI components mature. In our opinion, the most important efforts Sun can make to ensure the future success of Tcl and Tk are to (i) provide adequate support for object-oriented extensions to Tcl such as [incr Tcl], and (ii) to provide a seamless, efficient, and platform-independent interface to Java.

7 Implementation notes

7.1 Binary compatibility

One of the key factors in choosing implementation strategies for various features is that we want the Tycho core to run on any platform with no binary dependencies. To fulfill this requirement, we have made certain that Tycho will run within a standard **itkwish**. We have maintained a strong resistance to packages that require compilation to binaries, allowing this only if a) it is essential to gaining certain functionality and b) it is not going to significantly affect Tycho's cross-platform portability. The Tcl profiler from TclX [9], which we load into Tycho to work behind a graphical profile display, is a good example.

Another example of a C-coded package that adds useful (but not essential) functionality is support for integrating a Tycho-based custom user interface to a C program generated by Ptolemy. One of Ptolemy's features is an ability to generate C code from signal processing block diagrams. We have added a Tycho "target" to Ptolemy, which allows it to generate Tcl packages. Each package implements a simple execution protocol that allows us to load multiple packages into Tycho and interleave execution (while still keeping the user interface live). The screen shot in figure 1 show a real-time digital audio application running within Tycho – this ten-band stereo graphic equalizer easily runs at 44.1 kHz (CD quality) on a Sun UltraSparc.

A third example where we require binary support is to run Tycho with Java (section 6). Java is becoming more and more important in Tycho, however, and the need for a customized binary interface compromises our portability goals (despite the portability of the two languages separately). It is vitally important that Sun include a Java interface in the Tcl core.

7.2 Choice of widgets

Tycho uses its own library of mega-widgets. Tycho's widget library includes list and file browsers, font and color selection, alert boxes, an HTML displayer message window (for help messages), and so on. With most of these classes, we aim not to provide a complete "use-as-is" widget, but a foundation for customizing by inheritance and composition. For example, one of the key classes in the dialog box hierarchy is called *Query*: it contains a button box and an uncommitted frame for labeled entry widgets. A suite of methods add entry fields, check-buttons, option menus, and so on, to this frame. Many of the dialog classes – including the search dialog shown in figure 1 – inherit from this class and configure these fields in the constructor. This kind of extensibility is one of the great strengths of the object-oriented approach.

Throughout Tycho's development, we have faced the decision on whether to use existing widgets or to develop our own. The most suitable library is [incr widgets], or Iwidgets [16], a library of widgets written in [incr Tcl] that includes labeled widgets, scrolled canvas and text widgets, dialog boxes, a tabbed notebook, button and radio boxes, and so on. Other libraries, such as Tix and BLT, clashed with our "no-binaries" policy.

Tycho does not currently use any of the Iwidgets classes. We are not at all opposed to Iwidgets, and believe them to be a very useful set of widgets. In particular, we have used many ideas from Iwidgets and freely acknowledge our debt to the Iwidgets authors. In the future, widgets like the tabbed notebook may well be used within Tycho where we feel it is appropriate. When we began work on Tycho, however, Iwidgets was also early in development, and we found we had problems with many of them that were most easily fixed by writing our own. For example, moving the focus ring in the Iwidgets button box made the whole window adjust itself – visually, a very disconcerting effect. The Tycho button box widget does not do this.

Part of the reason that the Tycho widgets avoid some of the problems the Iwidgets authors faced is that they are somewhat more dedicated to our needs within the Tycho environment. We have not tried to deal with all possible uses, nor to provide all possible widgets. Instead, we have been able to choose what we felt was an adequate design where it was needed. Tycho widgets generally have far fewer configuration options than Iwidgets and more specific functionality. Iwidgets, for example, provides a scrolled canvas – the equivalent functionality in Tycho is a complete view with support for menus, printing, graphical selection, and so on.

There are secondary considerations that have worked against adopting Iwidgets. Tycho widgets benefit from our automatic documentation system, which includes executable Tcl examples embedded in HTML windows. Tycho widgets also appear on the Tycho class diagram, and are thus more visible to a Tycho developer.

7.3 Cross-platform porting

Our current release (May 97) is Tycho0.1.1. We are in the beta phase of Tycho0.2, which will be final in June 1997.

On the Macintosh, Tycho0.2 requires Itcl2.2p2. Under Itcl2.2 on the Macintosh, Tycho0.1.1 was crashing during startup. Unfortunately, inserting **puts** statements would move the location of the crash around. Jim Ingham, (formerly at Lucent, now at Sun) determined that the problem was that the Itcl2.2 Macintosh binaries had too small a stack to load all of Tycho. Under Itcl2.2p2, the Macintosh port starts up on some machines, but we were never able to complete startup on the infamous PowerBook 5300c.

We've tested Tycho0.1.1 and Tycho0.2 under NT, and most features work. The biggest problem is that non-blocking I/O is not available in the prebuilt Windows binaries. Under Windows, certain features of Tycho that start up subprocesses using non-blocking I/O do not work as they do under Unix. Unfortunately, the Macintosh does not even have **exec**, so the Macintosh port is even more limited.

A major issue facing the Macintosh and NT ports is the presence of spaces in file names. Passing filename arguments that contain spaces can be a little tricky. Support for file names with spaces could be improved – for example, the Tk4.2 demos that come with the prebuilt Itcl2.2 binaries do not work if there is a space in the pathname. Configuring Tcl8.0b under Unix also fails if there is a space in the prefix option to configure.

8 Concluding remarks

For user interface development, Tk is flexible and powerful. We have found that [incr Tcl] and [incr Tk] greatly improve the structure and clarity of medium-sized user-interface programs (Tycho is currently 64,000 lines of [incr Tcl]). Compared to the legacy Tcl/Tk code in Ptolemy, the Tycho code is marvelously well-organized. We would strongly recommend [incr Tcl] and [incr Tk] for Tk user-interface development for any substantial program.

[incr Tcl] and [incr Tk] are not without shortcomings. Many are largely inherited from Tcl/Tk: poor performance (which the new byte-compiler will greatly alleviate), and extremely lax parsing. In programs of this scale, run-time errors caused by an inadvertent extra parenthesis (for example), are extremely annoying and do nothing to give us confidence in the stability and robustness of our code. We have had problems with Tk's focus mechanism and the interaction of [incr Tcl] objects with Tk's update mechanism. As much as possible, we have worked around these, and other developers may wish to use the work-arounds incorporated into Tycho.

On the whole though, our experience in this project has been very positive and we are excited about future prospects. We urge Sun's Tcl/Tk group to focus on integrating Tcl/Tk (and [incr Tcl]/[incr Tk]) and Java into a seamless scripting/user-interface/code-development environment. This we see as the logical future of platform-independent computing.

Tycho is freely available under the unrestrictive UC Berkeley license – see the Tycho home page: `http://ptolemy.eecs.berkeley.edu/tycho/`.

Acknowledgments

Tycho is the work of many people. Contributors to the Tycho project include Kevin Chang, Wan-Teh Chang, Cliff Cordeiro, Wei-Jen Huang, Joel King, Farhana Sheikh, and Mario Jorge Silva. Key infrastructure without which this project would not have been possible has been developed by: Michael McLennan of Bell Labs ([incr Tcl]/[incr Tk]), John Ousterhout of U.C. Berkeley and Sun Microsystems (Tcl/Tk) Stephen Uhler of Sun Microsystems (HTML library), and Mark L. Ulferts of DSC Communications Corp ([incr Widgets]).

Tycho is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Dolby Laboratories, Hewlett Packard, Hitachi, Hughes Space and Communications, LG Electronics, Lockheed Martin ATL, NEC, Philips, and Rockwell.

References

[1] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4, April 1994. Special issue on Simulation Software Development.

[2] Dave Collins. *Designing Object-Oriented User Interfaces.* Benjamin/Cummings, 1995.

[3] Sven Delmas and Juergen Nickelsen. Information on TkSteal. `http://www.cimetrix.com/sven/tksteal.html`.

[4] Lisa Friendly. The design of distributed hyperlinked programming documentation. In *International Workshop on Hypermedia Design '95.* Sun Microsystems, Inc, 1995. `http://www.javasoft.com/doc/api_documentation.html#javadoc`.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reuse in Object Oriented Software.* Addison-Wesley, 1994.

[6] The Ptolemy Group. The Ptolemy home page. `http://ptolemy.eecs.berkeley.edu/`.

[7] The Ptolemy Group. The Tycho home page. `http://ptolemy.eecs.berkeley.edu/tycho/`.

[8] Edward A. Lee and David G. Messerschmitt et al. An overview of the Ptolemy project. `http://ptolemy.eecs.berkeley.edu/papers/overview/`, March 1994.

[9] Karl Lehenbauer and Mark Diekhans. The TclX distribution. `http://www.neosoft.com/tcl/ftparchive/TclX/`.

[10] Udi Manber, Sun Wu, and Burra Gopal. Glimpse: A tool to search entire file systems. `http://glimpse.cs.arizona.edu/`.

[11] Michael J. McLennan. The [incr Tcl] home page. `http://www.tcltk.com/itcl/`.

[12] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[13] Jan Nijtmans. Dashed and stippled outlines in Tk8.0a2 (Tk4.2p2, Itcl2.2). `http://www.cogsci.kun.nl/~nijtmans/tcl/patch.html`.

[14] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[15] Scott Stanton and Ken Corey. The TclJava demonstration. `ftp://ftp.sunlabs.com/pub/tcl/tcljava0.4.tar.gz`.

[16] Sue Yockey, Mark Ulferts, Bret Schuhmacher, John Sigler, and Alfredo Jahn. [incr Widgets]. `http://www.tcltk.com/iwidgets/index.html/`.

Building a Graphical Web History Using Tcl/Tk

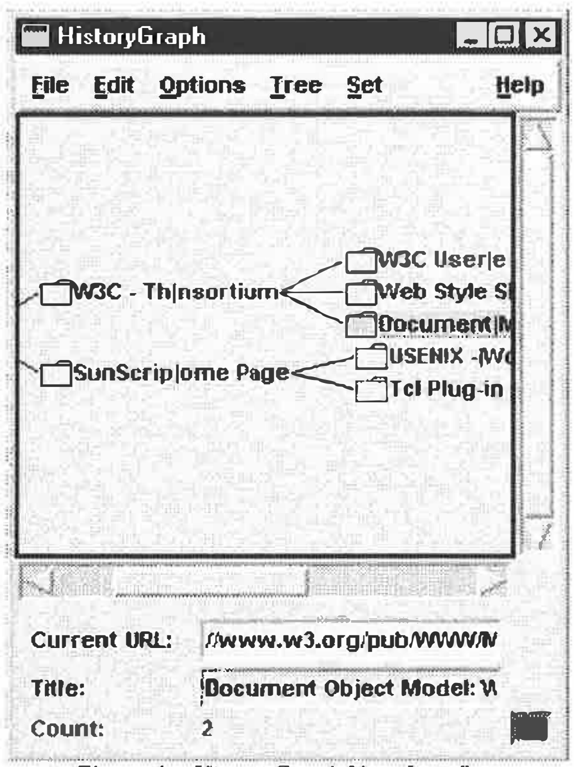
Frederick J. Hirsch (*f.hirsch@opengroup.org*)
The Open Group Research Institute
11 Cambridge Center, Cambridge MA 02142, USA

Abstract

This poster describes the design and implementation of a Web history tool that automatically tracks user browsing activities, presents a graphic visualization of this activity, and provides a mechanism for manipulation and use of the history. This tool, called *HistoryGraph*, demonstrates the power of using Tcl and Tk, especially through the reuse of existing components to create a powerful application in a short time.

Our goal was to create a browser independent tool which automatically creates a browsing history, making it easy to record sites visited, easy to return to sites, and possible to create trails to share with others. Unlike bookmarks, this history is displayed graphically, is directly manipulatable and may be integrated with other tools which work with URLs.

HistoryGraph automatically records the URLs and titles of sites visited, and builds a tree representation of the browsing history (See Figure 1). The user may reorganize this tree using drag and drop, and revisit sites by clicking on them in the tree. They may create sets of sites by selecting sites using pattern matches on the URLs or titles. Sets may be passed to other applications which may also add sites to the tree. The use of *HistoryGraph* is described in detail elsewhere [Hirsch97].



Current URL: //www.w3.org/pub/WWW/N
Title: Document Object Model: W
Count: 2

Figure 1 - *HistoryGraph* User Interface

We decided to implement *HistoryGraph* using Tcl/Tk in order to take advantage of the Tk graphical user interface toolkit, the rapid prototyping environment offered by Tcl/Tk, and the ease of integration with other web applications we had written in Tcl. Our group had attempted previous implementations of *HistoryGraph* using C and user interface libraries but these approaches had taken too long and were too difficult to modify. Using Tcl/Tk we were able to get a basic prototype operating in two months. Tcl/Tk also allowed us to perform incremental development, such as first creating a basic browsing history, then adding the ability to create and manipulate sets, and finally integrating *HistoryGraph* with other applications.

Implementation of *HistoryGraph* required more than Tcl/Tk code. We had to build special code for the NT platform, incorporate a tree drawing widget, and implement browser interfaces. Although Tcl/Tk is a portable interpreted language, we had to build an interpreter for Windows NT, and used the tknt40r1 package to do this.

In order to build an intuitive user interface, showing a graphical representation of sites visited, we used an existing tree widget. This tree widget was built in C++ and required building a special *wish* to incorporate it.

To make a browser-independent tool, we needed two application programming interfaces from browsers:

- An interface to install a callback which notifies the tool when the page changes, passing URL, title, and content to it.

- An interface for the tool to request that the browser load and display a page.

Not all browsers support these interfaces, but those which fully support the Spyglass Software Development Interface [SDI] do. Netscape Navigator 3.02 supports this on Windows NT using the Dynamic Data Exchange (DDE) implementation [Netscape-DDE]; Mosaic supports it on Unix using the NCSA Common Client Interface (CCI).

An alternative to using a browser interface is to intercept the HTTP stream between the browser and server, and capture URL and document information from the stream. This approach may become confusing to the user because the browser cache and "back" button may prevent all information from being transmitted over the network. This approach also requires the user to proxy their browser, and may not work with encrypted streams. We decided that close integration is best.

We integrated the *HistoryGraph* with other applications written in Tcl/Tk by using "send", and by later implementing a notification facility [Meeks]. One such application is designed to determine which pages have or have not changed since a specified time (WhatsNew) and the other is used to return the pages which are linked to by the page (LinkTree). Using Tcl/Tk allowed us to integrate these standalone applications in less than a week, substantially increasing the power of *History-Graph* through the reuse of existing applications.

Installation was an issue which made it difficult for users to use the system. Using *HistoryGraph* requires installing a special version of Tcl/Tk on their machine, installing *HistoryGraph*, and remembering to run this application when browsing. This is too cumbersome for everyday use. We would like to find an alternative implementation which would simplify installation and use. We are considering using the Tcl/Tk plugin which could simplify installation while allowing us to use the latest version of Tcl/Tk. This approach would require the use of shared libraries for custom functionality such as DDE support.

Our primary concern is the lack of a portable browser application programming interface which will continue to be supported. The only interface which supports installation of a callback for notification of page changes is DDE on NT with Netscape Navigator, or CCI on Unix with Mosaic. We have not discovered a way to implement this functionality using OLE, plugins, JavaScript or with Java Applets. General support for the Spyglass Software Development Interface is lacking, and full support for the DDE implementation is limited. Without such an interface, the techniques we used for *HistoryGraph* will be impossible to implement. What is needed is an portable application interface supporting tight integration of applications with browsers. Such a portable and supported interface would open the door to many interesting applications involving browsers. We need full support for an interface such as the Spyglass Software Development Interface, ideally from plugins.

A demonstration version of the original *HistoryGraph* software is available at
< http://www.osf.org/www/waiba/webwarev2_1/>

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under the contract number F19628-95-C-0042. The views and conclusion contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

My thanks go to Charles Brooks and Scott Meeks for their contributions to this project.

References

Brighton	A. Brighton, *Tree-4.0.1 - A Tree Widget for Tk4.0 based on C++ and [incr Tcl],* http://www.NeoSoft.com/tcl/ftparchive/alcatel/extensions/tree-4.0.1.tar.gz
Hirsch97	F. Hirsch, W. S. Meeks, C. Brooks, *Creating Custom Graphical Web Views Based on User Browsing History,* http://www.osf.org/www/waiba/papers/www6/hg.html
Meeks	W. S. Meeks, *Building a Notification Infrastructure Using Tcl, Zephyr, and Linda,* In this proceeding.
Netscape-DDE	Netscape's DDE Implementation, March 22, 1995. http://www.netscape.com/newsref/std/ddeapi.html
SDI	Software Development Interface, Spyglass, Inc. http://www.spyglass.com:4040/newtechnology/integration/iapi.htm
Welch	B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, 1995.

Building a Notification Infrastructure Using Tcl, Zephyr, and Linda

W. Scott Meeks
The Open Group Research Institute, 11 Cambridge Center, Cambridge MA 02142, USA

Poster Abstract

This poster describes the design and implementation of a notification system that can be integrated with other Web tools and applications. A Tcl-based toolkit using the MIT Zephyr Notification System was built to allow applications to notify each other or a user of relevant changes in information. A blackboard service was built using the notification toolkit and a server for the Linda language. This service is used to selectively store notifications for later use by applications. The use and effectiveness of the toolkit and service were demonstrated in an application for conducting a distributed guided tour of individual Web pages, and by integrating notifications from our Mediator service into our HistoryGraph application.

We tested our technology in two sample applications. The first was the GroupWalk program. It was written from scratch in Tcl to help in providing a "guided tour" of individual Web pages. This particular application provides a good example of the sort of cooperation and coordination we want to be able to support using the notification system.

When run, the program can take on one of two roles. In the first role, it watches the tour leader's browser and sends out a notification whenever the current URL changes. This information is also saved in the blackboard. In the second role, it receives these notifications and tells a tour member's browser to fetch the correct URL. Should a tour member arrive late for the tour, they can "catch up" by automatically retrieving (using information stored in the blackboard) all pages up to and including the current page in the order accessed (thus enabling sensible usage of the browser's *Back* button).

The second application was to integrate our Mediator [Mediator] service with our HistoryGraph [Hirsch97] application. The Mediator service provides controlled access to a set of structured documents for a team of users. It runs as an HTML forms-based CGI application. The HistoryGraph application tracks and displays browser navigation history in an easily manipulable form. It is written in Tcl/Tk. The main HistoryGraph display consists of nodes representing visited URLs and links representing the order in which URLs were visited.

Our primary goal was to provide a simple and extensible system for sending and receiving asynchronous event notifications between Web tools as well as users. We wanted to support dynamically changing groups of users and tools in a scaleable manner, and also wanted users and tools to only receive notifications of interest to them, so we chose a system that supports multicasting in something of a publish and subscribe fashion.

However, we also wanted to provide for persistent storage of certain notification information than can be easily shared among our tools. In essence, we wanted something like a blackboard that could serve both as a message repository and as a rendezvous service. Based on the mobility of both users and services (such as the blackboard) running on the network we desired that notifications be delivered independent of the actual location of the entity r eceiving it.

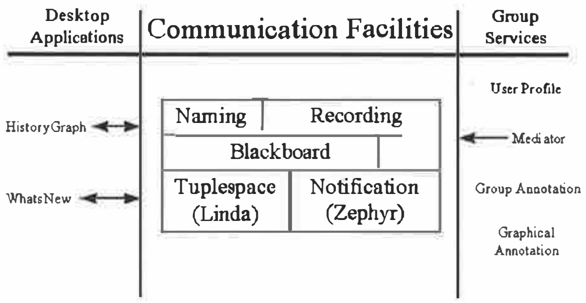


Figure 1: The notification software architecture and its use.

Figure 1 shows the relationship and structure of the notification software and our other applications and services. For the infrastructure of the notification system, we chose the MIT Zephyr Notification System [DelaFera87] and for the persistent store a server implementing the Linda[Schoenfeldinger95] language. We considered implementing our own infrastructure, perhaps by using Tcl-DP, but it made sense to use Zephyr since it provided a large body of existing code including the servers, host managers, zwrite program and the library API for accessing all of this. Likewise, we could have implemented the blackboard directly in Tcl, perhaps using ndbm, but since we already had the server which provided the persistent store as well as the Linda synchronization semantics, it made sense to use that.

In the Zephyr system, "multiple, redundant Zephyr services provide basic routing, queuing, and dispatching services to clients that communicate with the Zephyr client library."[DelaFera87] Essentially, Zephyr provides a location independent publish and subscribe model with a three-level notice classification scheme and optional authentication. Since these are all features we wanted to exploit and the Zephyr software is freely available, it made sense for us to incorporate it rather than reproduce it.

Linda is a small set of functions that can be added to a language to make a parallel dialect of that language. The functions are used to access a *tuplespace* which contains tuples of arbitrary information. We could have chosen other database solutions, but the Linda server was already written, was freely available, provided parallel programming and synchronization semantics that we hoped eventually to exploit, and was already being used in other Web applications.[Schoenfeldinger95]

There were a number of reasons we wanted to build a toolkit in Tcl, but primarily we liked the rapid prototyping capabilities and it provided compatibility with a number of our other applications and tools. However, we were particularly attracted by some of the new features in Tcl7.5.

Because of the different needs of our applications we wound up using at least half a dozen different versions of Tcl/Tk (including versions like Tcl-DP and IncrTcl as well as our own custom shells) amongst the various components of our software. The complexity involved made interoperability and maintenance difficult. Between the load command and packages, sockets, and the clock function in Tcl7.5, we were able to eliminate all but one specialized shell for the HistoryGraph application. After our experience with building the Zephyr package, we're fairly certain we could eventually eliminate that one.

The notification toolkit, blackboard, Mediator, and HistoryGraph are available at http://www.osf.org/www/waiba/webwarev2_1/. The GroupWalk associate was a very early prototype and a version for external release has not been pr oduced.

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under the contract number F19628-95-C-0042. The views and conclusion contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

My thanks go to Charles Brooks and Frederick Hirsch for critical readings of the original paper and this abstract.

References

[DelaFera87] C. A. DelaFera, *The Zephyr Notification System*, MIT Project Athena documents, Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.

[Hirsch97] F. Hirsch, W. S. Meeks, C. Brooks. *Creating Custom Graphical Web Views Based on User Browsing History*. http://www.osf.org/www/waiba/papers/www6/hg.html

[Mediator] *Distributed Authoring with the Mediator*. http://www.osf.org/RI/PubProjPgs/med-one.html

[Ousterhout94] J. K. Ousterhout, Tcl and the Tk Toolkit, *Addison-Wesley*, 1994.

[Schoenfeldinger95] W. Schoenfeldinger, WWW Meets Linda: Linda for Global WWW-Based Transaction Processing Systems. World Wide Web Journal, Issue 1: Conference Proceedings, Fourth International World Wide Web Conference, O'Reilly and Associates, December 1995. http://www.w3.org/pub/WWW/Journal/1/schoen.174/paper/174.html

[Welch95] B. B. Welch, Practical Programming in Tcl and Tk, *Prentice Hall*, 1995.

A Flexible GUI Design System

S.D.Mullerworth
The Meteorological Office
Bracknell, RG12 2SZ
UK
sdmullerworth@meto.gov.uk

The Generic Hierarchical User Interface (GHUI) is a design package for creating forms-based user interfaces and was written at the UK Meteorological Office. The intention is that the package is simple enough, that a basic, working interface can be written quickly by developers with little experience of the package, but that later, the interface can be enhanced by incorporating application specific code.

Generic functions provided by the GHUI include a client/server database system that allows users to save and copy work, a hierarchical tree widget for navigating with ease among a large number of windows, a flexible function for processing users' input into a text format that is suitable for controlling the related application, a function for creating uncomplicated input windows containing standard text, entry box, button and table widgets, and an error checking facility to prevent input of invalid responses. Application specific functions can be added to provide, for example, additional input validation checking or non-standard input windows.

A GHUI-based user-interface is specified by a set of text control files with a simple syntax. For example, each input panel, such as the one shown in the figure overleaf, is specified by a list of instructions in a single control file. Amongst other control files is a register that contains declarations for each of the data items set by the user, and a set of template files that define how the input to the application is to be converted to a format suitable for running the related application.

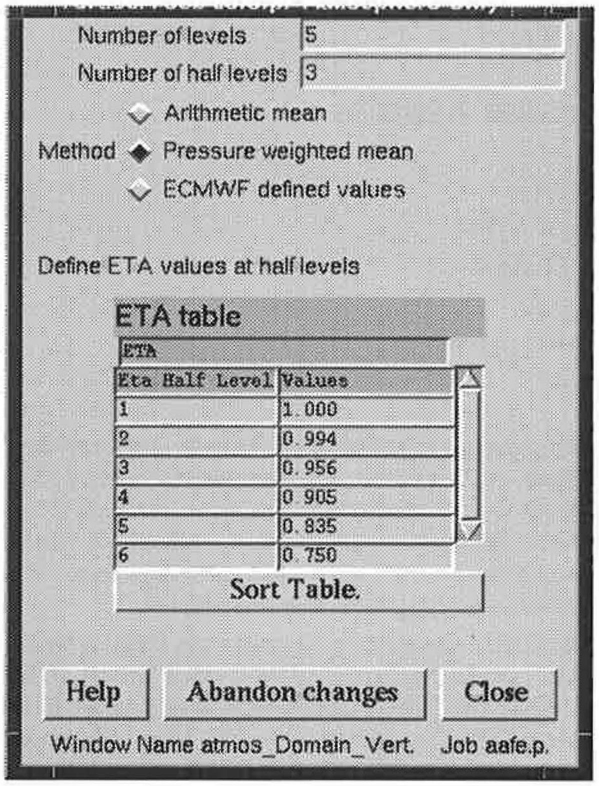
An important design aim was to ensure that each of the types of control file was easy to understand and therefore easy to write. The constraints that this design aim placed on the format of control files had an additional advantage in that it was then possible to write additional functions that reread the same control files for different purposes. For example, rather than using TclTk commands, input panel control files are written in a higher level GHUI language. Such an approach has enabled a number of useful generic functions to be written that base their output on these files; for example, a function that creates a text description of a user's settings. Such generic functions would be more difficult to implement if the use of TclTk commands to enhance the appearance of input panels was allowed.

The GHUI was written to create the Unified Model User Interface (UMUI). The Unified Model (UM) to which the UMUI interfaces is a very large, flexible modelling package used by the Met. Office both for its wide-ranging forecast products and for its climate prediction programmes. To offer the full flexibility of the UM to users, the UMUI requires some two hundred separate input windows. While professional looking applications have been developed using only the generic functions provided by the GHUI system, the UMUI takes full advantage of the opportunities for incorporating application specific code.

Much of the application specific code in the UMUI relates to the validation of user input. Each item in the database can have one of a standard set of checks applied to it, for example, to constrain the range of a numerical input. Alternatively, a specially written validation routine can be specified which enables complex conditions and cross-checks with other input to be applied. Tcl is an effective language for creating the required short scripts.

Another more substantial piece of application specific TclTk code provides the UMUI with a non-standard input window design in an area where the basic nature of the standard GHUI input panel was unsuitable. Although all the application specific code in the UMUI has been written in TclTk, clearly it would be possible to use other languages or incorporate input panels created by other GUI application builders.

All extensions are incorporated into the application in almost the same way as the standard GHUI functions; generally by listing the name of the function in the appropriate control file. Thus, the fact that


Number of levels 5

Number of half levels 3

Method Arithmetic mean [✓ Pressure weighted mean] ECMWF defined values

Define ETA values at half levels

ETA table

ETA

Eta Half Level	Values
1	1.000
2	0.994
3	0.956
4	0.905
5	0.835
6	0.750

Window Name atmos_Domain_Vert. Job aafe.p.

Figure 1: A simple example input panel from the UMUI showing entry boxes, a set of radiobuttons and a table widget. The three buttons at the bottom of the window are common to all panels. 'Help' brings up local help for the window, 'Close' and 'Abandon' close the window with or without saving changes.

a function is application specific is transparent to the user, yet such functions are relatively uncoupled from the GHUI system and they are thus unlikely to be affected by GHUI system upgrades.

The decision to design the GHUI rather than use an existing GUI application builder was made in part because of the need to use public domain software as it was intended that the UMUI was to be distributed free. Use of text control files has brought some other significant advantages when compared with many GUI application builders. Firstly, alterations can be done quickly with a text editor and no compilation or rebuilding process is required. Secondly, the files are readable by humans which is useful, for example, when searching for a particular question on one of 200 panels; simply search the input window control files for a particular string. Thirdly, again with regard to the number of windows required, the text format is much less cumbersome than the C code or resource files generated by many GUI packages.

Our experience demonstrates the feasibility of maintaining a suite of GHUI-based applications. Incorporation of application specific code to overcome the restrictions of the generic functions has not caused significant problems. The GHUI can provide the basis for user interfaces to a whole class of scientific packages. The GHUI approach, of designing an application to provide basic core functionality which can be enhanced with application specific code, could be applied to other classes of application where common requirements can be identified.

A World Wide Web-to-Database Connectivity using Tcl/Tk: The Hydrology-Meteorology Toolkit

François Chevenet
Guillaume Le Stum
ORSTOM, Hydrologie, BP 5045
911, Av. Agropolis
F-34032 Montpellier
chevenet@orstom.fr, lestum@orstom.fr

Abstract

The Hydrology-Meteorology Toolkit is a Tcl/Tk applets library for numerical/graphical processing hydro-meteorological data. It uses Tcl/Tk as a basic language. With the Tcl/Tk Plugin installed on the W3 browser side, applets use the HTTP protocol for querying an Oracle database server and managing i/o C programs towards Tcl CGI scripts.

1. The MED-HYCOS project

The World Meteorological Organization (WMO), with the support of the World Bank, promotes the development of a World Hydrological Cycle Observing System (WHYCOS), and the first regional WHYCOS component is the MED-HYCOS project (MEDiterranean HYdrological Cycle Observing System). ORSTOM (French Institute of Scientific Research for Development through Cooperation) hosts the MED-HYCOS Pilot Regional Center.

A MED-HYCOS aim is managing and numerical/graphical processing of hydro-meteorological data *via* the World Wide Web (W3).

2. The Hydro-Meteorology Toolkit

The Hydro-Meteorology Toolkit (HMT) is a W3 tools library for the MED-HYCOS project. HMT is based on a W3-to-Database connectivity using Tcl/Tk as a basic language.

An Oracle relational database management server deals with hydro-meteorological data such as streamflow discharges, temperature, rainfall, and so on.

On the W3 browser side (Navigator, Explorer), HMT needs the Plugin Tcl/Tk to be installed. A Tcl applets (tclets) library deals with data edition, graphical representations, exploratory data analysis (Tukey, 1977) and multivariate statistical analysis. At present, these tclets use the Trusted security policy.

Depending on user interaction, tclets use the HTTP protocol for (i), querying the database towards OraTcl CGI scripts. OraTcl is a well known Tcl extension from T. Poindexter that provides access to a Oracle Database server, and (ii), managing i/o C programs for multivariate analysis methods (*e.g.* Principal Components Analysis) towards Tcl CGI scripts.

Figure 1 shows two tclet examples.

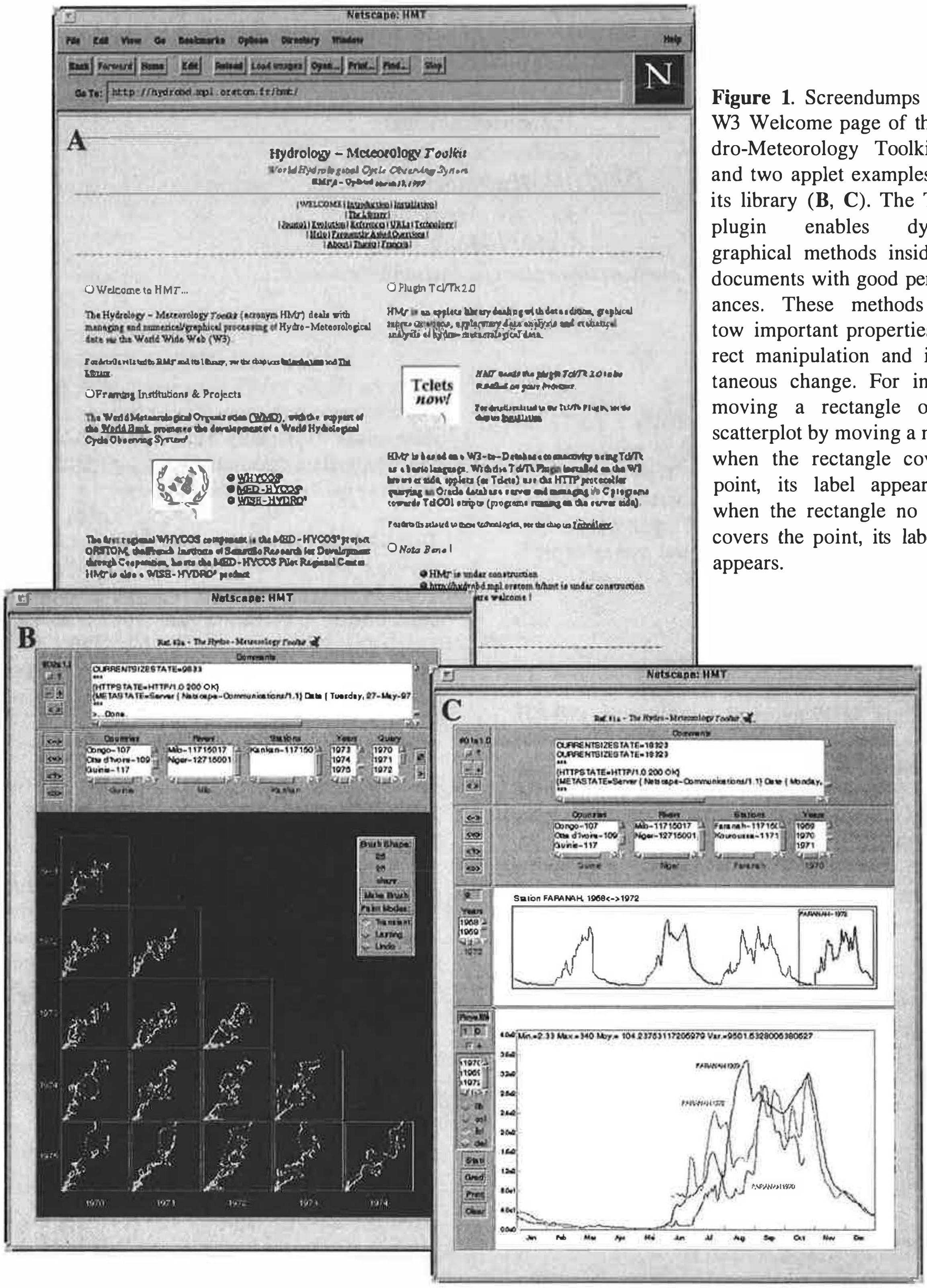
3. Conclusion

At present, HMT (http://hydrobd.mpl.orstom.fr/hmt) is still under development, but firsts results are decisive. The Tcl/Tk environment enables a powerful Human-Machine interaction inside W3 documents, such as dynamic graphical methods (Becker, Cleveland & all, 1987), with good performances. Moreover, it facilitates the development of the information system toward the homogenization of its architecture (*i.e.* Tcl for the CGI scripts instead of Perl and Pro*C for the SQL queries, and Tcl/Tk for applets).

References

Tukey J. (1977). *Exploratory Data Analysis.* Addisson-Wesley .

Becker R.A., Cleveland W.S. and A.R. Wilks (1987) Dynamic Graphics for Data Analysis. *Statistical science,* v2n4, 355-395.


Netscape: HMT
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Load Images Open... Print... Find... Stop
A
B
C

Go To: http://hydrobd.mpl.orstom.fr/hmt/

Hydrology – Meteorology Toolkit

World Hydrological Cycle Observing System

HMT/1 – Updated March 13, 1997

| WELCOME | Introduction | Installation |
| The Library |
Journal	Evolution	References	URLs	Technology
Help	Frequently Asked Questions
About	Thanks	Francais

Welcome to HMT...

The Hydrology – Meteorology Toolkit (acronym HMT) deals with managing and numerical/graphical processing of Hydro-Meteorological data via the World Wide Web (W3).

For details related to HMT and its library, see the chapters Introduction and The Library.

Framing Institutions & Projects

The World Meteorological Organization (WMO), with the support of the World Bank, promotes the development of a World Hydrological Cycle Observing System.

WHYCOS
MED-HYCOS
WISE-HYDRO'

The first regional WHYCOS component is the MED-HYCOS project. ORSTOM, the French Institute of Scientific Research for Development through Cooperation, hosts the MED-HYCOS Pilot Regional Centre. HMT is also a WISE-HYDRO' product.

Plugin Tcl/Tk 2.0

HMT is an applets library dealing with data edition, graphical [illegible], exploratory data analysis and statistical analysis of hydro-meteorological data.

Tclets now!

HMT needs the plugin Tcl/Tk 2.0 to be installed on your browser.

For details related to the Tcl/Tk Plugin, see the chapter Installation.

HMT is based on a W3-to-Database connectivity using Tcl/Tk as a basic language. With the Tcl/Tk Plugin installed on the W3 browser side, applets (or Tclets) use the HTTP protocol for querying an Oracle database server and managing its C programs towards TclCGI scripts (programs running on the server side).

For details related to these technologies, see the chapter Technology.

Nota Bene !

HMT is under construction
http://hydrobd.mpl.orstom.fr/hmt is under construction
... are welcome !

Ref. #1a – The Hydro-Meteorology Toolkit

Comments

```
CURRENTSIZESTATE=0833
***
{HTTPSTATE=HTTP/1.0 200 OK}
{METASTATE=Server { Netscape-Communications/1.1} Date { Tuesday, 27-May-97
***
>...Done.
```

Countries	Rivers	Stations	Years	Query
Congo-107	Milo-11716017	Kankan-117150	1973	1970
Cote d'Ivoire-109	Niger-12716001		1974	1971
Guinee-117			1975	1972
Guinee	Milo	Kankan		

Ref. #1a – The Hydro-Meteorology Toolkit

Comments

```
CURRENTSIZESTATE=10323
CURRENTSIZESTATE=10323
***
{HTTPSTATE=HTTP/1.0 200 OK}
{METASTATE=Server { Netscape-Communications/1.1} Date { Monday,
***
```

Countries	Rivers	Stations	Years
Congo-107	Milo-11716017	Faranah-11716	1969
Cote d'Ivoire-109	Niger-12716001	Kouroussa-1171	1970
Guinee-117			1971
Guinee	Niger	Faranah	1970

Station FARANAH, 1968<->1972
FARANAH-1972
Years
1968
1969
1972
Min.=2.33 Max.=340 Moy.= 104.23753117206979 Var.=9501.5328006380627
FARANAH1969
FARANAH1971
FARANAH1970
Stat
Grid
Print
Clear
Jan Feb Mar Apr Mai Jun Jul Aug Sep Oct Nov Dec

Figure 1. Screendumps of the W3 Welcome page of the Hydro-Meteorology Toolkit (**A**) and two applet examples from its library (**B**, **C**). The Tcl/Tk plugin enables dynamic graphical methods inside W3 documents with good performances. These methods have tow important properties : direct manipulation and instantaneous change. For instance moving a rectangle over a scatterplot by moving a mouse, when the rectangle covers a point, its label appears and when the rectangle no longer covers the point, its label disappears.

Using Tcl/Tk in Biology Research Application Development

Ellen R. Bergeman and Mark Graves

Department of Cell Biology, Baylor College of Medicine

One Baylor Plaza, Houston, TX 77030

phone: +1 (713) 798-8105; fax: +1(713) 798-3759

email: erb@bcm.tmc.edu

Biology research is an area which presents many problems for software application development. Biology application development occurs within a framework of rapidly changing needs; unspecified or incomplete requirements; short term versus long term needs; complex, often incompletely defined concepts; and multiple platforms with varying infrastructure. Biology researchers rely heavily on applications developed to analyze data, access databases, and store ongoing research data. Often the bottlenecks in research projects are not in the laboratory but instead are in the lack of software needed to handle the data.

Biology research changes rapidly and new laboratory techniques can be brought into a laboratory quickly, becoming integral elements in a very short time. Often a new technique is brought in because it will speed up the work in the lab, and therefore it changes the requirements for software support. For biology application developers, the need for an application is often not known until there is a bottleneck in the research process.

Another aspect of rapid change within biology research is that there are no clear-cut representations of the concepts within biology. The point of biology research is to explore unknown concepts to understand and clarify them. Application development should proceed in the same way as biology research so that applications are built to support current concepts and are allowed to evolve as research clarifies and changes the concepts.

We have found that Tcl and Tk are useful for solving some of the problems such as capturing complex concepts and improving communication between researcher and application developer. Complex concepts can be captured as data types which are initially developed in Tcl, then implemented in C as extensions to the Tcl language when they have been refined. Widgets which provide user-interface functionality for the data types can be implemented using TK. Using these data types and widgets allows for rapid development of small applications which can be used to improve communication between biologist and computer scientist.

A key to successful biology application development is having a framework in which a developer is able to try out different computer science solutions to the same problem without incurring substantial cost in development. Tcl and Tk provide a way for computer scientists to follow the same research paradigm as biologists: explore data representations for concepts as the concepts are evolving, while providing useful solutions to current software needs.

A major problem in biology software development is in the area of databases. It is difficult to design databases which capture the complex concepts and relationships of biology. Biology concepts have a graph-like structure, and we have developed a data model based on graphs which is useful for capturing biology data. Along with the data model, we have developed a graph database which stores and manipulates data as graphs. We have developed an extension to Tcl which includes the data types used in the graph database. These data types include an in-memory data storage graph and graph abstraction for complex queries. We have also developed widgets which provide graphical interfaces for the data types.

3wish: Distributed [incr Tcl] Extensions for Physical-World Interfaces

Brygg Ullmer
MIT Media Lab
20 Ames St., E15-445
Cambridge, MA 02139 USA
ullmer@media.mit.edu / http://www.media.mit.edu/~ullmer

Abstract

The creation of physical-world interfaces seamlessly integrated with the physical environment poses new implementation and interface challenges for the Tcl language. 3wish is a suite of [incr Tcl] class libraries and C/C++ extensions which supports user interfaces integrating distributed physical sensors, displays, and 3D graphics. The poster presents an overview of 3wish, its application to physical-world interfaces, and its implementation of distributed sensors, displays, object proxies, and platform-independent 3D graphics.

1. Introduction

Traditional 2D graphical user interfaces have been well-supported by the Tcl/Tk language. A new kind of user interface, "tangible user interfaces," uses physical objects and surfaces as physical interfaces to digital information. [1] These interfaces require sensing and augmentation by multiple physical-world sensors and displays, often distributed across several computers.

Tcl and the [incr Tcl] object/namespace extensions are well-suited for this style of user interface. Tcl's platform-independent, high-level operation, coupled with [incr Tcl]'s OOP class and namespace mechanisms, jointly support the rapid integration of access methods for distributed platform-specific sensors and displays. At the same time, new features are required to cleanly support physical-world interaction and distributed operation. 3wish is a suite of [incr Tcl] class libraries and C/C++ extensions that addresses these needs.

2. 3wish

3wish is designed to support user interfaces driven by physical objects. These physical objects must sense and process their physical state (position, orientation, etc.), coordinate among each other, and display various outputs (graphics, sound, etc.) to the user. Some of these sensors and displays are physically attached to interface objects; examples include position-trackers and flat-panel displays. Other sensing and display is performed on behalf of passive physical objects. For example, computer vision is used to track objects, while transparent objects are illuminated by back-projected displays. A sample interface is pictured in Figure 1 and discussed in detail in [3].

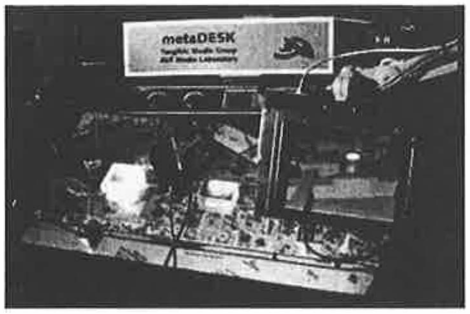


Figure 1: Image of metaDESK-based 3wish application

3wish hides these implementational details from the user interface. 3wish provides an [incr Tcl]-based "proxy" class for each physical object. [2, 3] These classes provide methods for querying object state and activating supported displays. However, underlying sensor/display coordination is hidden by the API. This abstraction supports a high degree of flexibility in constructing complex interfaces without becoming immersed in implementational minutia.

3. Distributed sensors and displays

At its lowest level, 3wish communicates with drivers for physical-world sensors and displays, currently operating on both SGI and Wintel platforms. Tcl 7.6's `load` command is used to load drivers for devices with C and C++ API's, where Tcl-DP 4.0 is used to communicate with serial devices.

Sensor APIs are implemented using an [incr Tcl] sensor class which provides caching and optional interpolation for sensor values. The same class is shared on both client and server machines; peer server and client classes work to synchronize sensor client and server states. Client sensor value requests are non-blocking, using the most recent cached/interpolated value available.

New sensor implementations usually require no additional networking code, as client/server code is abstracted from individual sensor fields and access methods. Similar display support is underway, though is currently at an earlier stage of development.

4. Object proxies and namespaces

Applications do not query sensor and display clients directly. Instead, [incr Tcl]-based "proxies" are provided as API's for each physical object. The resulting object-centric access methods are independent of the underlying sensing/display technologies employed. An illustration of this "proxy-distributed" or "proxdist" architecture is illustrated in Figure 2, and discussed further in [2] and [3]. Distributed namespaces are used to provide clean distributed coordination and avoid host/port/protocol dependencies.

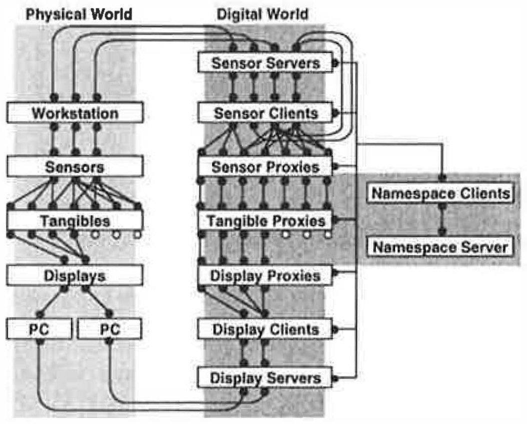


Figure 2: Diagram of 3wish sensor/display architecture

3D Graphics

A first iteration of 3wish developed a set of platform-independent 3D graphics extensions, inspired by the 2D Tk toolkit. 3wish's 3D graphics core draws on the Open Inventor (OIV) toolkit, using TGS's OIV port on non-SGI platforms. 3wish provides commands for asserting named OIV/VRML geometries, binding events involving these geometries to Tcl callbacks, etc. 3wish also supports registering scene graphs across multiple 2D and 3D displays, binding graphical geometries to physical objects, as well as providing other distributed graphics capabilities.

References

[1] Ishii, H., and Ullmer, B. "Tangible Bits: Towards Seamless Interfaces between People, Bits, and Atoms." In *Proc. Of CHI'97*, pp. 234-241.

[2] Ullmer, B. *Behavioral Realizations of Proxy-Distributed Computation.* http://www.media.mit.edu/~ullmer/-courses/agents/paper1.html March 1996.

[3] Ullmer, B., and Ishii, H. "The metaDESK: Models and Prototypes for Tangible User Interfaces." Submitted to UIST'97.

Using Tk as remote GUI frontend for 4GL-database applications

Volker Schubert
Brueckner&Jarosch Ing.GmbH
Erfurt, Germany, 99084
leo@bj-ig.de

Abstract

This is a short report about our experience using Tk as GUI frontend. We wrote a compiler called **F4GL** *, source compatible to the* **INFORMIX 4GL** *database language. (4GL means 4th generation language) This language was originally designed to create UNIX database applications with text-terminal output. 4GL programs compiled with* **F4GL** *generate Tcl-commands as graphic output (the text output is also still available). These Tcl-commands are sent over TCP/IP to a presentation server which will provide for conversation into graphic objects. We have presentation servers for X11 and Windows.*

4GL Server *is the presentation server for all Windows platforms and is implemented using Tcl/Tk. It is an Internet-Server like lpd or rshd whose services are available at a specific TCP port. We found different solutions for the network connection to the remote GUI / presentation server and had to solve various problems, which are discussed in detail in this poster. Writing a internet server in Tcl which evaluates Tcl-commands is very easy, but it introduces a security hole. The machine running* **4GL Server** *is accessible from the internet with tcl-commands. That's why we implemented a special protocol to authenticate the users of the* **4GL Server** .

*We made some extensions, some of it appears to be useful for the whole Tcl-community. Until now we maintained our own Windows-port of Tk3.6, because the actual Windows-Tk4.x releases from Sun still have limitations, we will explain the reasons and show some performance bottlenecks (***http://www.bj-ig.de/speed.html***). In the near future we want to switch to official releases because the Tk4.xx and Tk8.xx versions offer many advantages, especially safe interpreters and the Tcl-Plugin. A good overview about the* **F4GL** *can be found at* **http://www.4js.com/f4gldoc.html***, this is the webpage of* **4JS***, our trade-partner.*

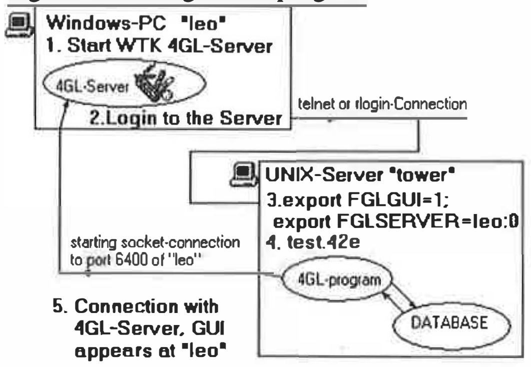
1 History

Two years ago we searched for a solution to migrate existing 4GL-database-programs to GUI's. First, we discussed a cross-compiler to translate 4GL into another database-language with builtin GUI (New Era, Powerbuilder, Gupta a.s.o) but this would exclude expierenced 4GL-programmers from the development. Therefore we decided to write a source compatible compiler.That produces programs with a configurable, flexible GUI-frontend. We choose Tk for the remote GUI, because we had some experience with it (usaging it for visualization in measurement tasks under Linux/X11), and it was free. The only problem was a good Windows port for the actual Tk3.6 version, because the majority of customers were expected using Windows (especially in Germany). We took one 16bit-port from **Software Research Associates Inc.** and started development at the Win32s platform to support native controls, sockets, optimize speed, and eliminate bugs. We call this Tk derivative **WTK** and it's of course freely available with all sources. See more information at **http://www.bj-ig.de/wtk.html**. Of course we want to switch now to the main distribution, to use all the very nice features of Tcl/Tk8.0, but there is one reason , which doesn't allow us to use it: performance.

2 Usage/Invocation

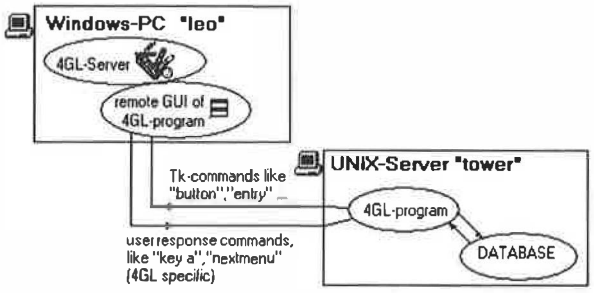
We'll begin with a short overview, of how to launch a database program, and how the communication between db-program and remote GUI is established (using Windows as frontend system). The user clicks on an icon, and a `rlogin` connection to the UNIX database host is established, the command line is automatically transferred, and the 4GL-program starts.
It then reverse connects to the Windows-machine (it reads an environment- variable `FGLSERVER` comparable with the `DISPLAY` variable for X11-applications) with our **4GL Server** at a specified port (6400) and transferes Tcl/Tk-commands like `button` or `entry` to the windows-machine. These commands are evaluated and

Figure 1: Starting a 4GL-program at the UNIX-Server



shape the GUI. Because the `rlogin` window can be kept hidden, the user images locally the program started.

Figure 2: Communication between 4GL-program and remote GUI


Windows-PC "leo"
4GL-Server
remote GUI of 4GL-program
UNIX-Server "tower"
Tk-commands like "button","entry" ...
4GL-program
user response commands, like "key a","nextmenu" (4GL specific)
DATABASE

If the user clicks on a button or writes to an entry, special bindings guarantee that this action is sent back over the socket as a simple text command.
The `rlogin` connections and the socket-connections of the 4GL-programs run in one application at the windows side, this is our product **4GL Server** ,completely written in Tcl (**WTK** =Tk3.6+some extensions).

3 Extensions

According to the frontend-character of the used widgets we made some extensions to Tk3.6. Because the run-time system of the database application controls all activities of the remote GUI, some widgets used have totally changed bindings as mentioned above.

3.1 The class-patch

The most important patch for us is the "class-patch" which allows the Tk-developer to give every widget another class, not only the `frame` and `toplevel` widget. This allows for example to create entries with different classes, each class can have other definitions of options in the option database. For each class there are other bindings possible. Both possibilities are often used in **4GL Server** . We already posted the patch to comp.lang.tcl, but it's unfortunately not contained in the actual distribution (one reason for writing the poster).

3.2 native controls

The main reason for developing **WTK** was to have a native look and feel of the GUI at the Windows-platform. The following controls are native windows controls under **WTK** : all kinds of `button`'s,`label`,`scrollbar`,`menu`,`term`. .

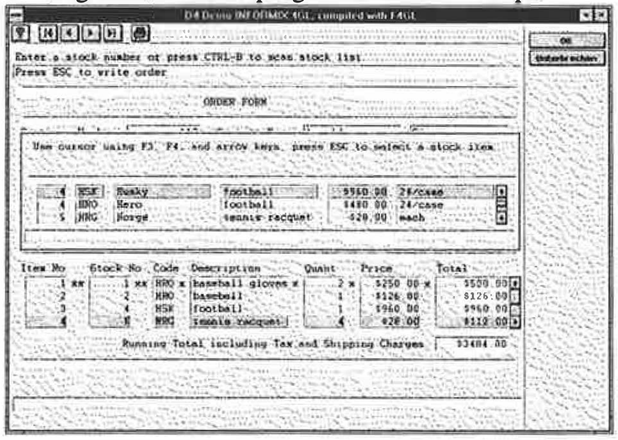
3.3 wtcp extension

wtcp-tcl is yet another socket implementation for Windows, it supports all kinds of tcp-sockets, asynchronous callbacks(WSAAsyncSelect) and winsock specific calls . Now the "socket"-command is available in the major distribution, and we want to switch, but not all functionality we need for **4GL Server** is included(setsockopt, transfer binary zero's for protocol- handling).

3.4 term extension

`term` is a terminal widget with configurable 3D pseudo-graphic symbols and configurable attributes. For example, the attribute bold you can assign a 3D-background color and a relief (comparable with Tun Terminal Emulation). `term` was used to implement a full-featured xterm Emulation in tcl, which is part of **4GL Server** . In the middle of the year we will introduce it for the tcl-community as a TK8.0-extension, but until now it's **WTK** -specific and only as native Windows widget available.

Figure 3: the 4GL-program D4 with tcl-output



```
Enter a stock number or press CTRL-B to scan stock list
Press ESC to write order

                    ORDER FORM

Use cursor using F3, F4, and arrow keys, press ESC to select a stock item

   4  HSK  Husky    football         $960.00  24/case
   4  HRO  Hero     football         $480.00  24/case
   5  HNG  Norge    tennis racquet    $20.00  each

Item No  Stock No  Code  Description       Quant  Price      Total
   1 **      1 **  HRO * baseball gloves *   2 *  $250.00 *  $500.00
   2         2     HRO   baseball            1    $126.00    $126.00
   3         4     HSK   football            1    $960.00    $960.00
   4         5     HNG   tennis racquet      4     $28.00    $112.00
          Running Total including Tax and Shipping Charges   $3484.00
```

xmb - dancing in the tar pits

Using Tcl/Tk to build a CASE environment

Henry R. Tumblin
CertCo L.L.C.
55 Broad Street
New York, NY 10004
tumblin@certco.com

Charles E. McElwain
Open Market, Inc.
245 First Street
Cambridge, MA 02142
mcelwain@openmarket.com

Managing software across multiple build platforms and environments can be thought of as dancing around a tar pit; if you slip and fall then you may not be able to get out for a long time! There's far too much knowledge about the local build environment required to enable software developers and release engineers to build, test, and release products. We intend to show that by using **Tcl/Tk** as a CASE integration engine, the amount of time and training required to bring a new developer, test or release engineer online is greatly reduced. Product build environments become standardized and the cost of adding new platforms is reduced.

The challenge we faced was one of release engineers having to build on multiple platforms, developers needing to build and test on a subset of these platforms, all working from a variety of individual environments and experience levels.

The problems that had to be solved were:
• distributing builds across the multiple platforms
• centralizing knowledge of build configurations in a tool (for repeatability)
• giving feedback on system resource usage and sharing
• providing better feedback and control (via a UI) and minimize training
• evolving a CASE tool towards greater maintainability, extensibility and reusability.
• adding multiple projects, and co-ordinating and analyzing inter-project dependencies

We looked at solving these problems through a three layer tool containing: a point-and-click GUI, a CASE kernel, and the remote tasks. The tool was based on a "build flow" which ran through:

Select Project -> Select Platform(s) -> Select Options -> Distribute -> Analyze

The problem was then how to turn this into a programmatic flow using **Tcl/Tk**.

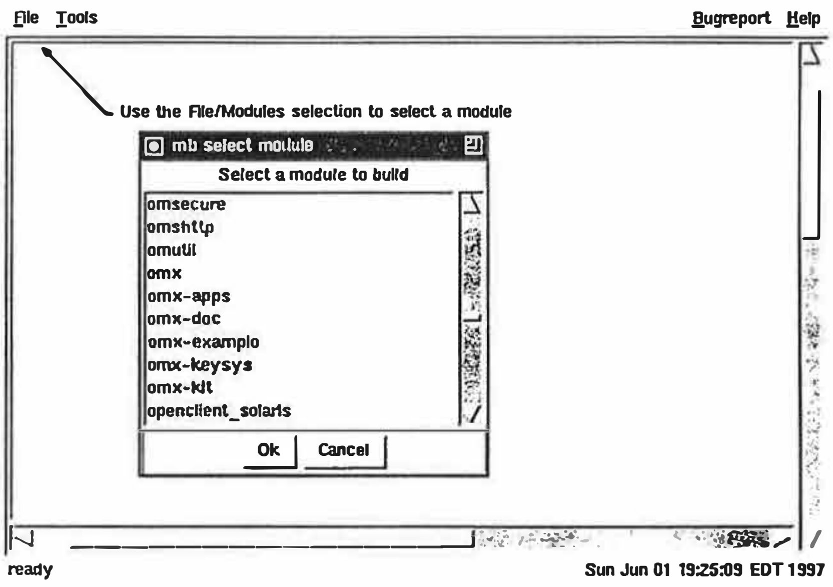
We built a small kernel representing these tasks, whose behavior is controlled by and changeable via config files which are themselves **Tcl**. Scripts to execute on the remote hosts were generated, then launched on the remote host. From within these scripts, host-specific config files were sourced to enforce the build environment and policies, as well as supplemental information, including compilers, compiler flags, paths, etc.

These scripts captured statistical information on each distributed build in the form of a **Tcl** statement representing a database entry. The reduction code for this is a small kernel which sources all database entries as text files and builds several structures as crosses of the data, and provides an API to access these views.

With the CASE kernel implemented, we then built a point-and-click GUI using **Tk**. We could now easily supplement the earlier **Tcl** kernel with point-and-click interfaces to the above stages of the build process. Control, feedback and configuration was now also visual. Training on the CASE environment became easier, intuitive and standard across multiple platforms and projects.

For maintainability of the CASE tool, we had to minimize global state. We did this by a hierarchical structuring of state data, class-oriented naming of procedures which contain self-initialization (constructor) code, along with standard module headers for extracting documentation.

In order to more easily support and analyze multiple projects, both within single invocations and across multiple invocations of the CASE tool, we used the same techniques as in handling the need to report resource usage via a **Tcl**-based database. A daemon was then implemented to co-ordinate and serialize writing to the database, utilizing TCP primitives in **Tcl** (thus avoiding the need for custom socket code).



Use the File/Modules selection to select a module

Select a module to build

omsecure
omshttp
omutil
omx
omx-apps
omx-doc
omx-example
omx-keysys
omx-kit
openclient_solaris

ready

Sun Jun 01 19:25:09 EDT 1997

Figure 1: Selecting a project or module to work on. This list is generated from the underlying source control system, CVS in this case.

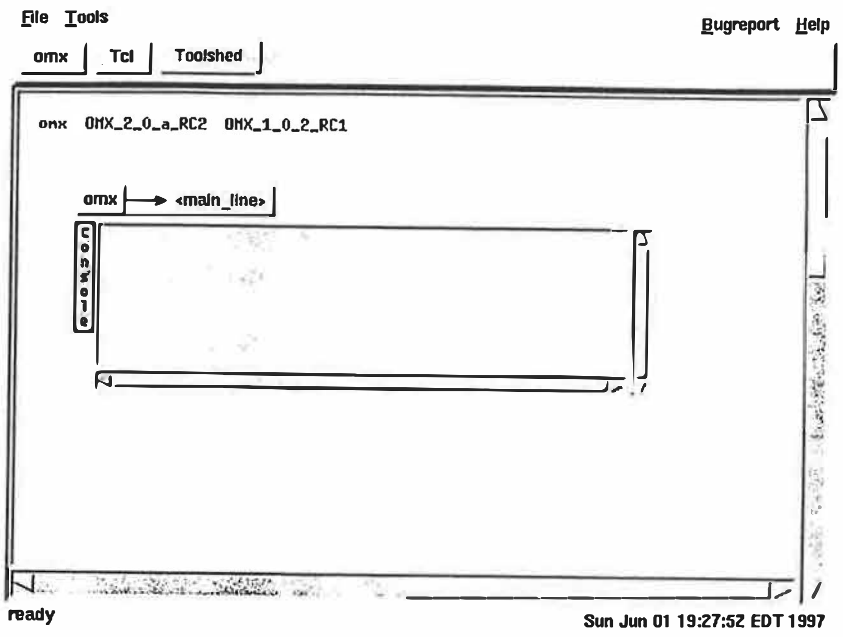
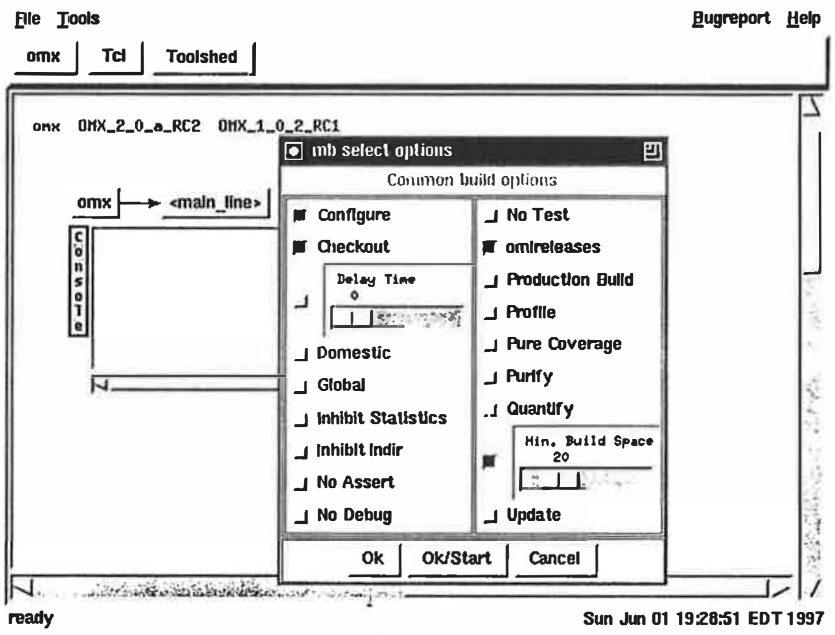
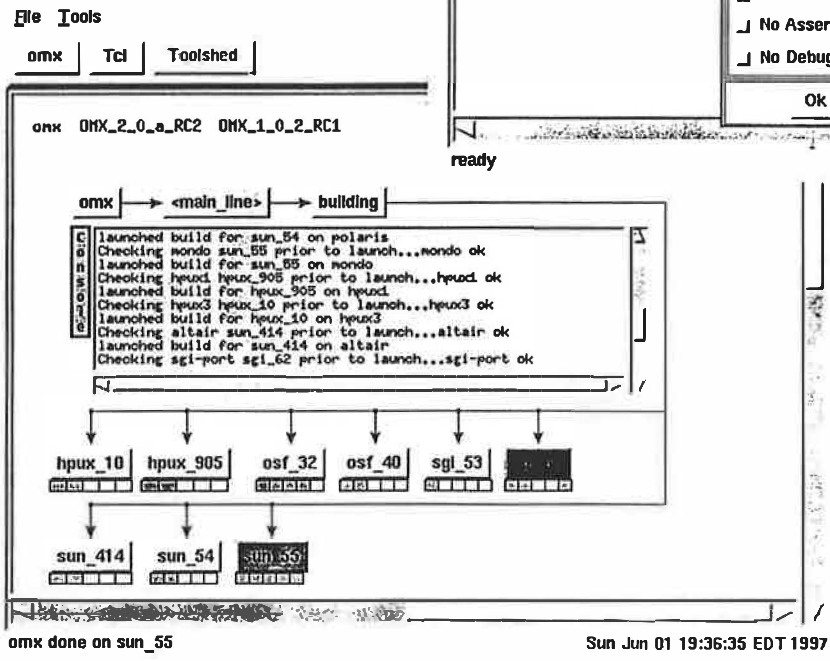


Figure 2: Selecting different projects, **omx**, **Tcl**, and **Toolshed**, and multiple tags, *main_line*, *OMX_2_0_a_RC2*, *OMX_1_0_2_RC1*, *within one of them*, (**omx**).



■ Configure
■ Checkout
❑ Delay Time 0
❑ Domestic
❑ Global
❑ Inhibit Statistics
❑ Inhibit Indir
❑ No Assert
❑ No Debug
❑ No Test
■ omireleases
❑ Production Build
❑ Profile
❑ Pure Coverage
❑ Purify
❑ Quantify
■ Min. Build Space 20
❑ Update

Figure 3: Selecting build options for the current project **omx** and tag **mainline**.



```
launched build for sun_54 on polaris
Checking mondo sun_55 prior to launch...mondo ok
launched build for sun_55 on mondo
Checking hpux1 hpux_905 prior to launch...hpux1 ok
launched build for hpux_905 on hpux1
Checking hpux3 hpux_10 prior to launch...hpux3 ok
launched build for hpux_10 on hpux3
Checking altair sun_414 prior to launch...altair ok
launched build for sun_414 on altair
Checking sgi-port sgi_62 prior to launch...sgi-port ok
```

Figure 4: Watching a build in progress across multiple platforms. One has succeeded, *sun_55*, one has failed, *sgi_62*, and the others are still in progress.

Agent Development Support for Tcl

R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Tim Finin, Ethan Miller and Charles Nicholas
Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland 21250
{rcost1, ian, jlakha1, finin, elm, nicholas}@cs.umbc.edu

In the past few years, the explosive growth of the Internet has allowed the construction of "virtual" systems containing hundreds or thousands of individual, relatively inexpensive computers. The agent paradigm is well-suited for this environment because it is based on distributed autonomous computation. Although the definition of a software agent varies widely, some common features are present in most definitions of agents. Agents should be autonomous, operating independently of their creator(s). Agents should have the ability to move freely about the Internet. Agents should be able to adapt readily to new information and changes in their environment. Finally, agents should be able to communicate at a high level, in order to facilitate coordination and cooperation among groups of agents. These aspects of agency provide a dynamic framework for the design of distributed systems.

Tcl is an ideal language with which to build agents, because scripts written in Tcl may be used on any machine that can run Tcl, and because the Tcl language environment itself is highly portable. Additionally, Tcl/Tk greatly facilitates rapid prototyping and quick development of small applications.

We present TKQML, the integration of an agent communication language, KQML [6] (Knowledge Query Manipulation Language) into Tcl/Tk. TKQML can be used to build KQML-speaking agents that run within a TKQML shell. TKQML can also be used to bind together diverse applications into a distributed framework, using KQML as a communication language. Tcl's embeddable nature allows one to easily add agent communication facilities to existing code. As such, TKQML can be used to enhance the functionality of new or existing systems built using a Tcl framework, by allowing easy integration with agent-based systems.

KQML is a language for general agent communication. It was developed as part of the Knowledge Sharing Effort [7], a DARPA project exploring agent communication and knowledge reuse. KQML is a language based on speech acts, such as "tell", "ask", and "deny", which describe the nature of a message without reference to its content. Agents communicate application-specific information embedded in general, higher-level KQML messages. A comprehensive semantics [4] for KQML outlines protocols for agent "conversation." Additionally, most implementations provide facilities for message handling, agent naming and resource brokering.

Problems of software mobility, communication, and autonomy have not been neglected within the Tcl community. Existing Tcl-based solutions to agent issues, such as AgentTcl [2] and Tacoma [3] have emphasized security and mobility, but fall short with respect to communication. AgentTcl agents, using TCP/IP, exchange bytes strings which have no predefined syntax. In Tacoma, agents must meet in order to communicate. Others projects, such as Tcl-DP [9, 5] provide excellent packages for communication, but lack sufficiently flexible support for higher level languages. TKQML bridges this gap.

The CARROT project (Co-operative Agent-based Routing and Retrieval of Text, formerly CAFE) is an ongoing effort at UMBC to develop a distributed architecture for text-based information retrieval [1], and has served as a testbed for TKQML. This project employs a brokered environment of clients and servers. Users make queries through a World-Wide Web-based client, which are routed intelligently by a broker agent to an appropriate information source. The broker makes these decisions by gathering information, or metadata, from each source, and deciding which database the query most resembles. A ranked set of results is returned to the client.

A heterogeneous set of text-indexing engines, such as Telltale [8] and mg [10] manage large sets of text data. These engines have been augmented into agents with TKQML. The broker agent communicates transparently with these information servers via KQML. All components consist of C/C++ applications bound to TKQML with a Tcl/Tk shell. One agent, the Agent Control Agent (ACA) is written entirely as a TKQML script. Our experience with CARROT has shown that TKQML can facilitate quick prototyping and rapid development of agents and their GUIs, reducing the time necessary to build large agent-based systems. Figure 1 presents a sample system, in which agents of varying types communicate via KQML.

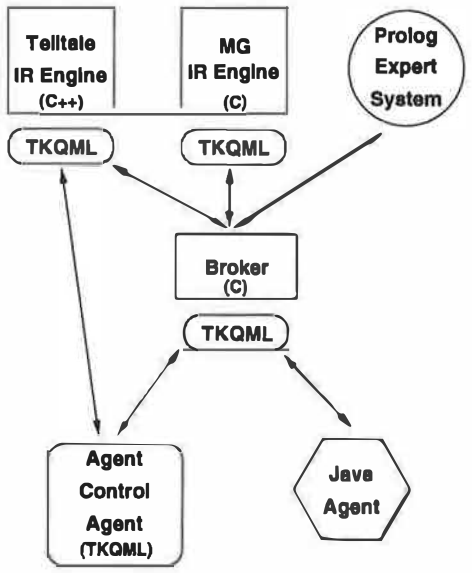


Figure 1: A system of agents speaking KQML. Some entities have been augmented with TKQML, while other have native KQML abilities.

Both Tcl and KQML are powerful tools in the development of agent-based systems. TKQML combines the two, making it possible to benefit from both the light weight and portability of Tcl scripts and the high-level communication support of KQML with one package. We feel that its power, simplicity and potential for future development make it an ideal platform for the development of agent-based systems.

References

[1] Grace Crowder and Charles Nicholas. Resource selection in CAFE: An architecture for network information retrieval. In *Proceedings of the Network Information Retrieval Workshop, SIGIR 96*, August 1996.

[2] Robert Gray. Agent Tcl: A flexible and secure mobile-agent system. In *The Fourth Annual Tcl/Tk Workshop Proceedings*. The USENIX Association, 1996.

[3] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An introduction to the TACOMA distributed system. Technical report, University of Tromso, June 1995.

[4] Yannis Labrou. *Semantics for an Agent Communication Language*. PhD thesis, University of Maryland Baltimore County, 1996.

[5] Peter Liu, Brian C. Smith, and Lawrence A. Rowe. Tcl-DP name server. In *Proceedings of the 1995 Tcl/Tk Workshop*. The USENIX Association, July 1995.

[6] James Mayfield, Yannis Labrou, and Tim Finin. Evaluation of KQML as an agent communication language. In J. P. Woolridge and M. Tambe, editors, *Intelligent Agents Volume II – Proceedings of the 1995 Workshop on Agent Theories, Architectures and Languages*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

[7] Ramesh S. Patil, Richard E. Fikes, Peter F. Patel-Schneider, Don Mckay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA knowledge sharing effort: Progress report. In Charles Rich Bernhard Nebel and William Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR92)*. Morgan Kaufman, 1992.

[8] Claudia Pearce and Charles Nicholas. TELLTALE: Experiments in a dynamic hypertext environment for degraded and multilingual data. *Journal of the American Society for Information Science*, April 1996.

[9] Brian C. Smith, Lawrence A. Rowe, and Stephen C. Yen. Tcl distributed programming. In *Proceedings of the 1993 Tcl/Tk Workshop*. The USENIX Association, June 1993.

[10] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, 1994.

TxRx : An ONC RPC Interpreter

Cristian S. Mata*
Department of Computer Science
State University of New York
Stony Brook NY 11794-4400
cristian@cs.sunysb.edu

1 Introduction

TxRx is a run-time environment for ONC RPC tightly integrated with the Tcl language. Open Network Computing Remote Procedure Call [RFC1831] is a widely supported interprocess communication protocol currently on the standards track of the IETF. The eXternal Data Representation (XDR) is the language used by ONC RPC to describe inter-application communication. *TxRx* provides an interpreter for XDR and support for the features of ONC RPC.
TxRx improves the process of building distributed applications by eliminating most of the steps required to develop such a program. *TxRx* makes Tcl scripts compatible with existing client-server applications and facilitates access to system services like NIS and NFS.
RPC uses the *procedure call* abstraction to model the process of sending a request to, and receiving a reply from, a remote computer. The message sent to the remote is an encoding of the parameters of the procedure. The return value from the procedure call is the reply message received from the server. *TxRx* is intended for use in instances where compatibility with existing systems is important e.g. when existing client-server systems need to be upgraded.

2 Implementation

TxRx is a Tcl dynamically loadable package. It consists of a compiler/interpreter for XDR and code that implements RPC communication. A developer defines the communication protocol between two applications by specifying the remote procedures, parameters and return values in XDR. The next step – at run-time – is to load the protocol description using a *TxRx* command. The file with the protocol description is compiled by *TxRx* into bytecode.
In ONC RPC the internal data structures used by the protocol – the RPC headers – are defined using XDR. *TxRx* takes advantage of this feature by using a data driven approach: RPC headers are bytecode-compiled with the user protocol. When a message arrives from the remote computer, the incoming data stream is parsed according to the instructions stored in the bytecode. Conversely, outgoing data, including RPC headers, are converted from Tcl data into binary data by interpreting the bytecode program.
TxRx is layered, with different abstractions implemented at each level. The base level is the XDR interpreter. Its function is to create and manage objects that encapsulate the communication protocol. The level above it implements remote procedure call functionality and handles data buffer management, timeout and network transport semantics. The user level deals with authentication and security issues.
RPC processing in *TxRx* is independent of the communication channel. One advantage is that connection setup between client and server is done separately from data transfer.

3 Experimental results

The goal of *TxRx* is to reduce development time and improve application portability. The idea behind the experiments is that local processing times are small compared to network latency and throughput. The typical workload consists of RPC calls with variable payload size with little processing on both the server and the client. The time performance of *TxRx* is compared to the performance of a C program generated with *rpcgen*. All experiments were executed on Sun workstations running Solaris. The data in Figure 1 reflects the difference in speed between *TxRx* and C code. The times are the av-

*Supported by NSF grants CCR-9201585 and CCR-9501192 and by a grant from Hughes Aircraft.

erage times in seconds required to execute a given number of iterations. The client and server are on the same local network. The C code implementation is about 4 times faster than *TxRx*.
Figures 2 and 3 graph the ratio of times between *TxRx* and C clients with respect to the number of RPC calls respectively data transfer size between client and server. Somewhat surprisingly, there are no major differences in behavior when client and server are located on the same computer – label "Local" – or on the same Ethernet segment – label "Ether". The increase in the ratio of execution times can be attributed to the interpreted approach – Figure 2 – of *TxRx*. When client and server are hosts on the Internet – one host in *edu*, the other in *com* – performance becomes a function of overall data traffic on the Internet.
Figure 3 is the time ratio between *TxRx* and C code with respect to the size of the data transfer. In this case the key to understanding the graph is the structure of the transferred data – in this case a linked list. The performance ratio between *TxRx* and C-code decreases when the complexity of the data structure increases.

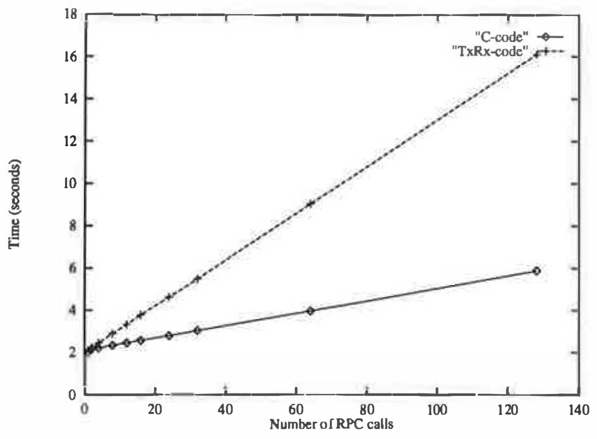


Figure 1: Execution time as a function of the number of calls

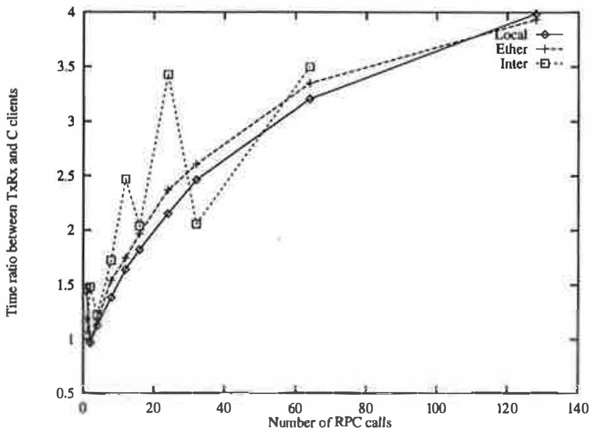


Figure 2: Performance ratio vs. number of calls

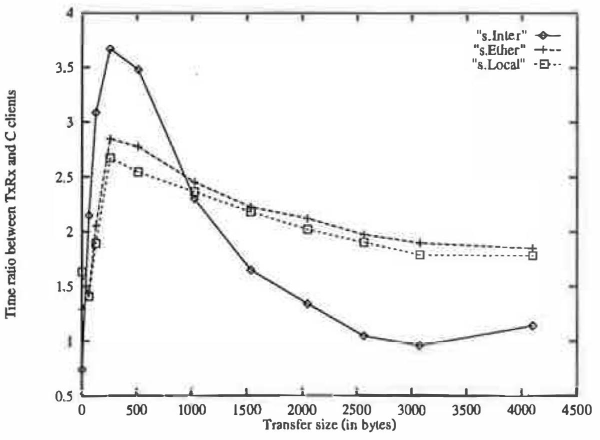


Figure 3: Performance ratio vs. transfer size

4 What next?

TxRx is currently implemented as a "C code + Tcl script" [TxRx] extension. This is for convenience and for efficiency reasons. As Tcl evolves – with the introduction of the bytecode compiler and native binary data handling in version 8 – it becomes possible to implement *TxRx* as a script only extension. Efficiency aside, this makes interesting applications possible, like a WebNFS [RFC2055] client coded entirely in Tcl.

References

[RFC2055] Brent Callaghan. SUN Microsystems, Inc. WebNFS Server Specification, October 1996. http://www.sun.com/webnfs/

[RFC1831] Raj Srinivasan. SUN Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification, Version 2. Request for Comments 1831, Standards Track, August 1995. http://ds.internic.net/rfc/rfc1831.txt.

[TxRx] Cristian Mata. SUNY Stony Brook. *TxRx*: An ONC RPC extension for Tcl. http://www.cs.sunysb.edu/~cristian/txrx.html

A Tcl/Tk-Based Video Annotation Engine

M. Carrer, L. Ligresti, and T.D.C. Little
Multimedia Communications Laboratory
Department of Electrical and Computer Engineering
Boston University, Boston, Massachusetts 02215, USA
tdcl@bu.edu

The population of a video database requires tools for manipulation and annotation of raw video data. Characteristic of this requirement is the need to satisfy many disparate video-based application domains. In this extended abstract we describe the design and development of a video annotation engine called Vane (Fig. 1), intended to address the issue of domain-independent video annotation. Rather than relying on a single, general data model and application interface, we developed a dynamic interface and data model using the Tcl/Tk environment and SGML document type definitions (DTDs). This approach allowed us to implement an intuitive graphical user interface application that is easily portable to different systems. The outcome of our work is a multipurpose, domain-independent video annotation application that has been developed taking advantage of Tcl/Tk features for easy construction and reconfiguring of GUI widgets at execution time. Thereby offering a novel application model appropriate for the domain.

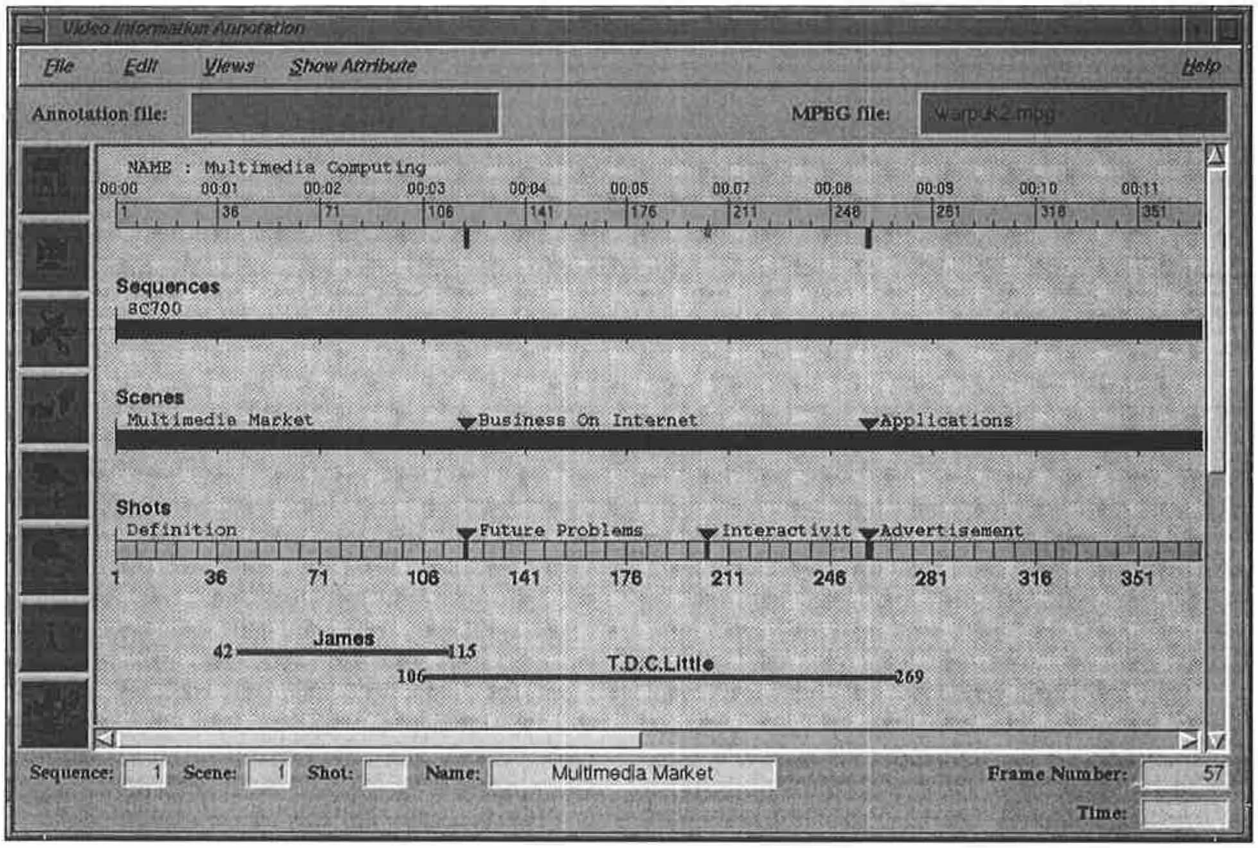
We built Vane so that the DTD can be modified in many of its parts to suit the needs of the annotator and to better describe the current domain under analysis. Opening a new annotation in Vane means identifying its associated DTD. The particular DTD is then parsed by a Tcl procedure, resulting in the loading of its syntax rules into an array in memory. When the annotation of one of the defined SGML elements is requested by the annotator, a new, top-level window is built (Fig. 2). Attributes belonging to these elements are mapped to a Tcl/Tk widget according to their type. These can be pop-up menus, form entries, listboxes, or text areas for attributes such as content transcripts. Defining a new DTD for a new annotation, or changing part of an existing DTD, does not affect the implementation of Vane, rather, these changes are accommodated by the construction of the user windows from the DTD on-the-fly.

Once the syntax of the annotation document has been extracted from a DTD, each of its fields is associated with its semantics from a corresponding SGML document. The latter carries the video information as annotation data. An additional Tcl procedure handles the output produced by an SGML parser by loading the field values into an annotation array. Because the array indices are based on the DTD array, and therefore on the DTD syntax, identification of format mismatches is easily accommodated. The reverse process, writing-out annotation metadata to SGML, is therefore straightforward as well. In this case we generate SGML output by following the syntax rules stored in the DTD array. The result is a "generated" SGML document which is certainly consistent with its own DTD.

With respect to the static user interface of Vane (Fig. 1), the canvas widget and its associated properties were used extensively. Each element appearing in the main window represents a "hot-spot" with an associated Tcl/Tk binding. Because the binding is based on a motion event, a simple dragging of the mouse pointer over an annotation element automatically updates other information boxes in the window. This offers to the annotator a set of easily accessible shortcuts to check the progress of the annotation process.

In summary, we have used Tcl/Tk as a scripting language and toolkit for the implementation of our annotation tool. The outcome of our work is Vane, a multipurpose, domain-independent video annotation application. It has been developed to achieve a dynamic, run-time-reconfigurable user interface by taking advantage of the unique characteristics of Tcl/Tk. The tool is currently in use to annotate a video archive comprised of educational and news video content in the Multimedia Communications Lab.

Additional details of the Vane implementation can be found at the following URL: *http://hulk.bu.edu/pubs/papers/1996/carrer-vane96/TR-08-15-96.html.*



Annotation file:

MPEG file: warpuk2.mpg

NAME : Multimedia Computing

SC700

Multimedia Market ▼Business On Internet ▼Applications

Definition ▼Future Problems ▼Interactivit ▼Advertisement

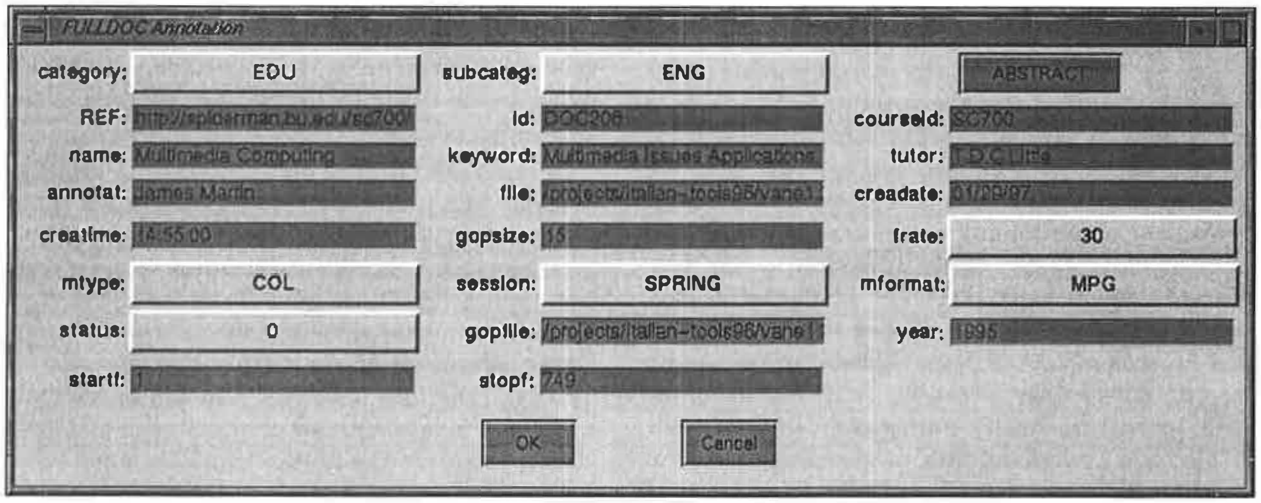
42 James 115

106 T.D.C.Little 269

Sequence: 1 Scene: 1 Shot: Name: Multimedia Market Frame Number: 57

Time:

Figure 1: Screenshot of the Vane Workspace Window



category:	EDU	subcateg:	ENG		
REF:	http://spiderman.bu.edu/sc700/	id:	DOC206	courseid:	SC700
name:	Multimedia Computing	keyword:	Multimedia Issues Applications	tutor:	T.D.C.Little
annotat:	James Martin	file:	/projects/italian-tools96/vane1	creadate:	01/28/97
creatime:	14:55:00	gopsize:	15	frate:	30
mtype:	COL	session:	SPRING	mformat:	MPG
status:	0	gopfile:	/projects/italian-tools96/vane1	year:	1995
startf:	1	stopf:	749		

Figure 2: Screenshot of a Vane Annotation Window

Coding Techniques for Reducing Code Maintenance

Clifton Flynt
Flynt Consulting Services

Abstract

Tcl/Tk supports code constructs that can reduce the amount of code that needs to be changed when code is modified.

The following examples and brief discussion explain some code conventions that can reduce code maintenance.

Shell or C programmers are familiar with using a switch statement to parse a command line. In Tcl the command line parsing code can reformat the arguments into Tcl commands. This allows new command line arguments to be added without code modification.

One such convention is "-varName value", which can be parsed by a set of code resembling example 1.

The names of global variables can be placed in a list to be evaluated by procs which need access to them. This creates a single point of change when new variables need to be declared global. Example 2 shows sample code.

Menu construction can be data driven instead of code driven. For example, a program which processes the contents of files can have the file selection menu built with the Tcl `glob` command. This allows new files to be automatically included in the menu. The traditional method of hardcoding a list of items to place in a menu would require a code modification whenever new files are added (or a new BaseDirectory is selected from the command line). See Example 3.

Saving and restoring state arrays can also be be data driven instead of code driven. Tcl can report all of the indices of an array. This list can then be used to drive the code which saves these values. By saving the variables as a Tcl command string, the state can be restored with the source command. See Example 4.

Further examples and discussion are included in the posters.

These conventions were developed while writing, extending and maintaining TclTutor.tk which is available at:

`http://www.msen.com/~clif.`

Clif Flynt can be reached as *clif@clif.ypsi.mi.us*

Example 1.

```
# Scan command line for -Varname Value combinations

for {set i 0} {$i < [llength $argv]} {incr i} {
  set arg [lindex $argv $i]
  if {[string first "-" $arg] == 0} {
    incr i;
    eval [list set [string range $arg 1 end] [lindex $argv $i]]
    } else {
    puts "Bad argument: $arg"
    }
  }
```

Example 2.

```
# Global variables are all declared in a single location

set globalList [list errorCode errorInfo stateArray argv argc]

proc GenericProc {} {
  global $globalList; eval "global $globalList";
  ...
  }
```

Example 3.

```
# Add a list of files to a menu

foreach file [glob $stateArray(BaseDirectory)] {
  $menu add command -label $file -command "ProcessFile $file"
  }
```

Example 4.

```
# Save state as a source-able file.

puts $stateFile "array set stateArray [array get stateArray]"
```

InvenTcl: Interpretive 3D graphics using Open Inventor and Tcl/[incr Tcl]

Sidney Fels, Silvio Esser, Armin Bruderlin and Kenji Mase,
ATR MI&C Research Laboratories
Seika-cho, Soraku-gun, Kyoto, 619-02, JAPAN
fels@mic.atr.co.jp   +81 774 95 1448   fax: +81 774 95 1408

Abstract

Open Inventor is an object oriented 3D graphics toolkit written in C++. Because Open Inventor is written in C++, typical user code development consists of a program/compile/debug iteration cycle. This paper introduces *InvenTcl* which is an interpretive version of Open Inventor using Tcl/Tk [4] and [incr Tcl] [3]. The advantages of InvenTcl include: script-able and direct manipulation of 3D objects in an Open Inventor scene, easy prototyping of 3D graphics and animation, and low-bandwidth communication of 3D scenes and animations (using scripts).

Discussion

There are three command types provided by InvenTcl which correspond to the main functions available in Open Inventor: object creation commands, object interaction commands and animation commands. For object creation there are command names for instantiating objects. These commands have the same name as the class names in Open Inventor, e.g. in InvenTcl there is a command called SoSeparator[1] which creates a new separator [incr Tcl] object with corresponding methods to access the public methods defined in Open Inventor for an SoSeparator. For interaction, InvenTcl has binding mechanisms to allow Tcl procedures to be called when objects in the 3D scene are selected. For animation commands, InvenTcl provides access to animation functions found in Open Inventor (i.e. engines and sensors). To illustrate creation commands with an example, the following code shows how a simple scene graph is created interpretively:

```
SoSeparator >root>separator1
SoMaterial >root>separator1>material1
SoCube >root>separator1>myCube
```

This series of commands adds an SoSeparator node *separator1* to the root node, an *SoMaterial* node to separator1 and an *SoCube* node to separator1. These commands coupled with other Open Inventor commands would display a cube with the material properties specified by material1.

[1]An SoSeparator is a common object used in Open Inventor programming. It is used to isolate the effects of nodes in a group from other nodes in a scene graph.

Notice, that we use the '>' notation to specify parent/child relationships.

There are four main technical issues to deal with to make Open Inventor interpretive:

1. accessing Open Inventor's object functions and the object's public methods and variables from the Tcl interpreter,
2. Open Inventor event management within Tcl/Tk,
3. binding Open Inventor objects to Tcl procedures and interaction modes,
4. synchronization of Open Inventor and Tcl processing.

To implement access to Open Inventor objects from Tcl we write command *wrappers* around the C++ instantiation functions for the Open Inventor library classes. Thus, a Tcl command is created with the same name as an Open Inventor class for instantiation. To access public variables such as fields, we provide command-line options like width, length, or radius. Alternatively, fields can be set after creation analogous to the Tk *configure* commands using Inventor's callback functionality with the appropriate *ClientData* pointer set. Our current implementation does not work effectively with the inheritance relationships specified in the Open Inventor class libraries and has commands which are tailored to our project [1]. In particular, we do not have convenient access to all the public methods for our classes. To remedy this we plan to use a utility called Itcl++[2] to convert Open Inventor object's methods into [incr Tcl] classes. Itcl++ is a utility which converts C++ class hierarchies into [incr Tcl]. It provides access to public methods inside each class and preserves the inheritance hierarchy. In [2], 32 classes and 190 member functions from the Open Inventor library were converted to [incr Tcl]. Using this utility we plan to convert the remainder of the class hierarchy of Open Inventor and integrate it with our notation, event handling, binding and synchronization techniques (see below). On top of the class hierarchy created with Itcl++, we will add the operator '>' for specifying parent/child relationships. Having this operator is useful for integrating Open Inventor scene graph descriptions with path specifications in Tcl/Tk. Itcl++ does not provide access to any public variables. We are currently addressing this issue by altering our approach for providing configure style access to public variables to work with Itcl++.

An Open Inventor event handler is installed as an asynchronous handler in Tcl to manage any Open Inventor events. Implementation of event management within Tcl does cause some performance penalty. Here is the Open Inventor event manager we use:

```
XtInputMask m;
XEvent event;
XtAppContext t;
t = SoXt::getAppContext();
if( m = XtAppPending(t ) ) {
  if ( m & XtIMXEvent ){
    SoXt::nextEvent(t, &event);
    SoXt::dispatchEvent(&event);
  } else   {
     XtAppProcessEvent( t, m );
  }
}
```

The main role of this handler is to find events which are related to Open Inventor and dispatch them to Open Inventor.

The main interaction command in InvenTcl is the **Ibind** command. The **Ibind** command allows users to bind an event and a callback procedure to objects in the Open Inventor scene graph. When used with objects in the scene graph this command is analogous to the canvas widget bind method for binding to objects on the canvas. However, our current version of InvenTcl allows only one scene graph and the **Ibind** command implicitly binds to objects in the one and only scene graph. Thus its syntax resembles the Tk bind command even though the objects that are bound are more like 3D versions of the canvas widget. Further, we do not provide support for Tcl binding to some of the interaction buttons and sliders that are provided by Open Inventor. Here is an example of the **Ibind** command:

```
Ibind >root>player>p5 <Ctrl-Button-1-Up>
      {puts "here"}
```

The binding mechanism is implemented using a combination of the event callback mechanism provided in Open Inventor and a Tcl_HashTable addressed by each object in the scene graph which is bound. Each object (referenced by a path in the scene graph) in the hash table has a structure associated with it to keep track of the bound callback function and any user call back data. In the example above, the path specified is >root>player>p5 and the callback function is {puts "here"}. The event which triggers the callback is a <Ctrl-Button-1-Up> event. When an event occurs, i.e. <Ctrl-Button-1-Up>, an Open Inventor callback node in the scene graph calls a generic callback handler. This generic callback handler looks in the hash table to see if the scene graph path which was selected is in it. If so, the generic callback gets the associated structure from the hash table and calls the bound callback function with the user data.

Synchronization between autonomous event driven Open Inventor activities and Tcl/Tk is performed using a global semaphore.

Summary

In summary, to date, the following is working:

- Implementation of a small subset of the Open Inventor library with limited access to public variables and methods.
- Integration of event management of Open Inventor and Tcl/Tk events.
- Implementation of a 3D binding mechanism allowing selection of 3D objects to call Tcl procedures.
- Implementation of synchronization between Open Inventor and Tcl/Tk.

Our current work is focussed on enhancing Itcl++ to translate all of Open Inventor's class library. To do this, we are modifying Itcl++ to allow access to public variables. Further, the classes created using Itcl++ are being integrated with our mechanisms for event handling, binding and synchronization. Once complete, we plan to use InvenTcl to create a 3D mega-widget canvas using the Tk [4] canvas widget as a model for design.

[1] BRUDERLIN, A., FELS, S., ESSER, S., AND MASE, K. Hierarchical agent interface for animation. In *Animated Interface Agents IJCAI workshop, to appear in Workshop Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'97)* (August 1997).

[2] HEIDRICH, W., AND SLUSALLEK, P. Automatic generation of Tcl bindings for C and C++ libraries. In *Proc. of the Tcl/Tk Workshop* (July 1995).

[3] MCLENNAN, M. [incr Tcl]: Object-oriented programming in Tcl. In *Proc. 1st Tcl/Tk Workshop* (University of Berkeley, CA, USA, 1993).

[4] OUSTERHOUT, J. K. *Tcl and the Tk Toolkit.* Addison-Wesley, New York, 1994.

Experience of Prototyping Tcl/tk-based GUI for Geant4; an Object-Oriented Toolkit for Simulation in High Energy Physics

M. Nagamatsu, [*] H. Uno, A. Obana, H. Yoshida [†]
Naruto University of Education
Naruto-shi 772, Japan
M.Asai
Hiroshima Institute of Technology
Saeki-Ku, Hiroshima 731-51, Japan
Y.Oohata, R. Hamatsu
Tokyo Metropolitan University
1-1 Minamioosawa Hachioji Tokyo, Japan

Abstract

This paper reports our experience of prototyping Tcl/tk-based Graphical User Interface for object-oriented GEANT4 toolkit. The menus and windows of the GUI are required to be automatically modifiable corresponding to various customization of GEANT4 by application programmers.

Without demanding of application programmers any knowledge on GUI programming, we provide them with formal methods to register their command and associated parameters. The back-end interface of GEANT4 converts the registered ones into messages and sends them to the Tcl/tk GUI script, G4LogTerm. It, in turn, parses the messages and creates corresponding hierarchical menus and parameter widgets.

Design choice of message passing method between OO-GEANT4 kernel and GUI rather than a monolithic solution using C++ GUI builder gives our GUI more flexibility and expandablility. One examplary direction is a GUI for network-distributed computation. Another is a platform independent GUI, since GEANT4 kernel is already so and can be run not only on Unix but also on personal computers.

The lesson we have learned in developing a non-freezing GUI for a long-running simulation program like GEANT4 is the importance of prototyping of execution control at system call level. The integrated programming environment of Tcl/tk including low level functions such as `fileevent`, `fconfigure` as well as its useful functions like list processing helped us to implement the key parts of the GUI.

[*]nagamatu@naruto-u.ac.jp
[†]yoshidah@naruto-u.ac.jp

References

[GEANT3] GEANT - Detector Description and Simulation Tool, CERN Program Library W5013

[GEANT4] A. Dell'Aqua et al., GEANT4: an Object-Oriented toolkit for simulation in HEP, CERN/DRDC/94-29, 1994

[Amako94] K. Amako: Object-Oriented Analysis and Design of GEANT Based Detector Simulator, CHEP94, San Francisco, 1994

[Olsen92] D. R. Olsen Jr.: User Interface Management System: Models and Algorithms, Morgan Kaufman, 1992

[Geant4 97] Geant4 Home Page: http://wwwinfo.cern.ch/asd/geant/geant4.html

[Ousterhout95] J. K. Ousterhout : Tcl & Tk Toolkit (Japanese language edition) SOFTBANK Corp., 1995

Web Enabling Applications

Brent Welch
Stephen Uhler
{bwelch,suhler}@eng.sun.com
Sun Microsystems Laboratories
2550 Garcia Ave. MS UMTV29-232
Mountain View, CA 94043

Abstract

This demo shows a Tcl-based Web server that can be extended with application-specific modules to support a variety of applications. The server is written as a pure Tcl script[Ouster94][Welch95] that is modular, extensible, and embeddable. Clients use a regular Web browser to make HTTP requests to the Tcl server. The server can be directly embedded into existing applications, or it can act as a stand-alone server that includes application-specific modules. The server web-enables your application so it can be accessed through a standard Web browser.

The architecture of the server allows interception of URL requests at two levels. At the top-level, the application can implement a whole sub-tree of the URL hierarchy. In this case the application is free to implement whatever semantics it wants for those URLs. The server provides an `Direct` module that works at this level. It maps URL requests directly into calls on Tcl procedures. The name of the URL determines the name of the Tcl command, and query data from forms is bound to procedure arguments. The return value of the Tcl procedure is returned as HTML data to be displayed in the browser.

For example, the URL `/mail/bugreport` is bound to `Mail_/mail/bugreport`, and this procedure sends its form data to the registered webmaster for the server. Note that it is not possible to invoke arbitrary Tcl commands in the server via this interface.

The second level of URL processing supports handlers for different document types. This level is supported by the `Doc` module that implements file-based URLs. The file name suffix is used to select a MIME content type [MIME93], and handlers can be defined for different content types. This level is used to support standard CGI, server-side includes, and imagemaps. Applications can define new file content types and register handlers by defining appropriately named Tcl commands.

The `application/x-tcl-subst` content type is used for hybrid HTML+Tcl documents. These pages are HTML templates that include in-line Tcl commands. The server processes the templates in Safe-Tcl interpreters. Side effects from the Tcl processing control the application, and the results are inserted into the Web page viewed on the clients. HTML forms allow input, and specialized displays can use Tclets or Java applets.

The server supports a series of page accesses by the same client, which are called a session. A session is identified by query data posted with the URL. Each session maps to an instance of a Safe-Tcl interpreter. The HTML+Tcl templates are processed in the interpreter for the session. Creating a session creates an interpreter and installs application-specific aliases. The aliases are intended to be the primary Tcl commands used in the HTML+Tcl templates.

Sessions are further subdivided into groups (for lack of a better name) that collect session state information. A group maps to a Tcl array within the per-session interpreter. A group may be associated with a page or shared by different pages, and a page could have several groups. Query data can be bound to groups, automatically updating the Tcl arrays based on query data.

The server supports other standard features like logging and authentication mechanisms. These are implemented by modules that can be tuned for application-specific needs.

The system includes an authoring environment for the HTML+Tcl templates that is based on the WebTk Tcl/Tk HTML editor [Welch96]. Content developers are presented with a simple property sheet interface to aliases that are used in the templates. They can develop the surrounding HTML in a WYSIWYG environment and easily add and tune the calls to the aliases. Open APIs are used to create the aliases and integrate them into the authoring environment.

The demo features SNMP device management [SNMP] via a Web browser. The server uses the Scotty SNMP Tcl extension to access and control SNMP-based network devices [Schoenwaelder95]. This application uses the HTML+Tcl templates. It defines a content type, `application/x-snmp`, that processes the query data in an SNMP-specific way and then uses the regular template mechanism. It uses sessions to record which device is being accessed. Groups within a session are used to control the data-driven SNMP module. SNMP uses a database, or MIB, that describes the device. Session groups are used on different pages to record what part of the MIB is being displayed and how it is displayed. The application includes a general MIB browser that lets you navigate the MIB to examine different parts of the SNMP state of the device.

The Tcl Web server is designed to be a flexible mechanism that facilitates web-access to new and existing applications. It leverages the embeddability and extensibility of Tcl so it is easy to add the server to your application. The architecture supports a variety of ways to map URL accesses to the application. In particular, the HTML+Tcl templates for dynamic page generation provide a rich infrastructure for building the web-based interface to your application.

Availability

The server is available for evaluation purposes. We hope to have a freely available version available soon. Please contact the authors for more information, or check the web site at `http://sunscript.sun.com/products/tclhttpd/`.

References

MIME93 RFC-1521. MIME (Multipurpose Internet Mail Extensions). N. Borenstein.

Ousterhout94 J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, April 1994, ISBN 0-201-63337-X.

Schoenwaelder95 Schoenwaelder, Langendorfer, *Tcl Extensions for Network Management Applications*. Proc. of the 3rd Tcl/Tk Workshop. ftp://ftp.ibr.cs.tu-bs.de/pub/local/papers/tcltk-95.ps.gz.

SNMP RFC 1155, RFC 1157. Simple Network Management Protocol.

Welch95 B. Welch, *Practical Programming in Tcl and Tk*. Prentice Hall, June 1997 (2nd Ed.), ISBN 0-13-616830-2.

Welch96 B. Welch and S. Uhler. *Tcl/Tk HTML Tools*, Proc. olf the 4th Tcl/Tk Workshop. July 1996. pp 173-182.

NOTES

NOTES

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of engineers, scientists, and technicians working on the cutting edge of the computing world. The USENIX technical symposia and system administrator conferences are the essential meeting grounds for the presentation and discussion of the most advanced information on the developments of all aspects of computing systems.

USENIX and its members are dedicated to:
- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's bi-monthly newsletter featuring technical articles, system administration tips and techniques, SAGE News, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as nine technical meetings every year.
- Discounts on the purchase of proceedings from USENIX conferences and symposia and other technical publications.
- Discount on purchases of USENIX CD-ROMs.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Nolo Press, O'Reilly & Associates, Prentice Hall, Sage Science Press, and John Wiley & Sons.
- Special subscription rates on periodicals: *The Linux Journal,* UniForum's *IT Solutions*, and the annual *UniForum Open Systems Products Directory.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Adobe Systems Inc.
Advanced Resources
ANDATACO
Andrew Consortium
Apunix Computer Services
Boeing Commercial
Crosswind Technologies, Inc.
Earthlink Network, Inc.
ISG Technologies, Inc.
Matsushita Electric Industrial Co., Ltd.
Motorola Research & Development
MTI Technology Corporation
O'Reilly & Associates
Sybase, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

Sage Supporting Members:

Atlantic Systems Group
Bluestone, Inc.
Enterprise Systems Management Corp.
Great Circle Associates
OnLine Staffing
Paranet, Inc.
Pencom Systems Administration/PSA
Southwestern Bell
Taos Mountain
Texas Instruments, Inc.
TransQuest Technologies, Inc.

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA. Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org.* URL: *http://www.usenix.org.*

ISBN 1-880446-87-1